

Calcolatori Elettronici T  
Ingegneria Informatica

# ISA DLX: implementazione *pipelined*



# Principio del *Pipelining*

Il *pipelining* è oggi la principale tecnica di base impiegata per rendere “veloce” una CPU .

L’idea alla base del *pipelining* è generale, e trova applicazione in molteplici settori dell’industria (linee di produzione, oleodotti ...)

Un sistema,  $S$ , deve eseguire  $N$  volte un’attività  $A$ :

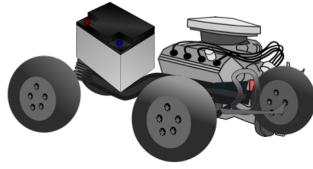
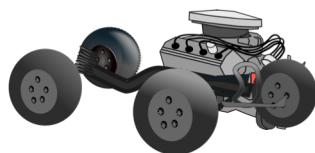


**Latency** : tempo che intercorre fra l’ inizio ed il completamento dell’ attività  $A$  ( $T_A$  ).

**Throughput** : frequenza con cui vengono completeate le attività.



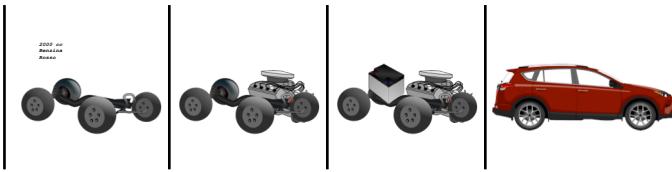
Motore: 2000 cc  
Tipo: Benzina  
Colore: Rosso



Motors  
Tires  
Chassis



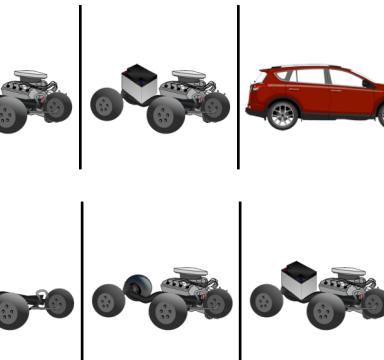
2000 cc  
Benzine  
Engine



Motors  
Tires  
Chassis



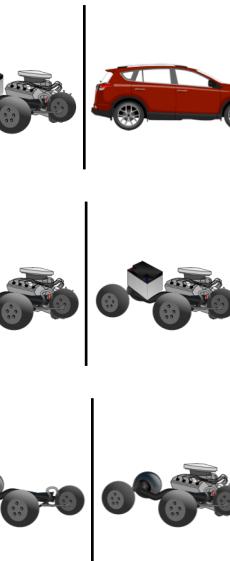
2000 cc  
Benzine  
Engine



Motors  
Tires  
Chassis



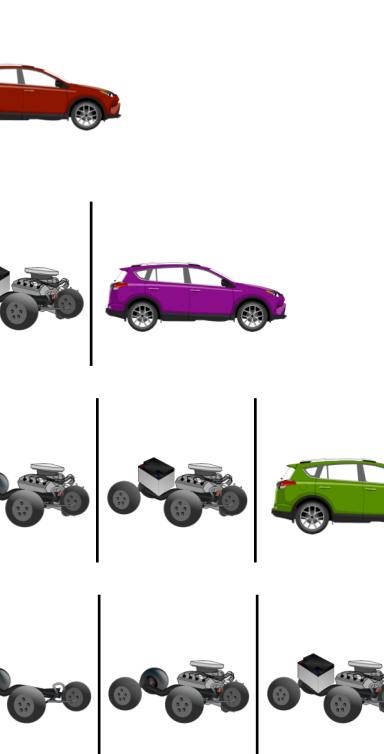
2000 cc  
Benzine  
Engine



Motors  
Tires  
Chassis



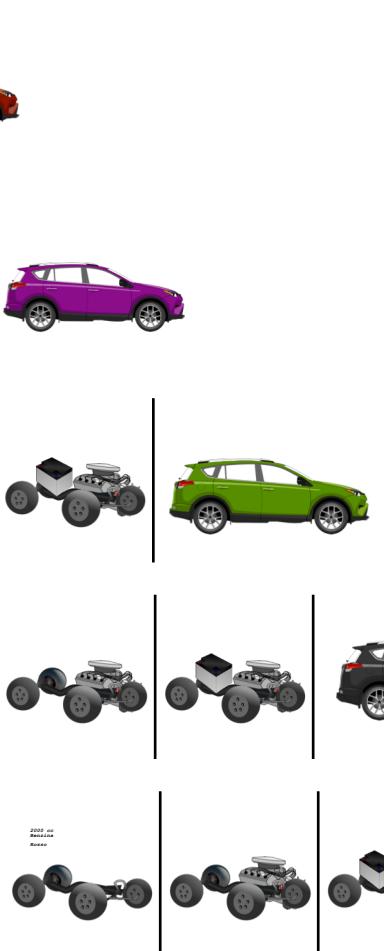
2000 cc  
Benzine  
Engine



Motors  
Tires  
Chassis



2000 cc  
Benzine  
Engine



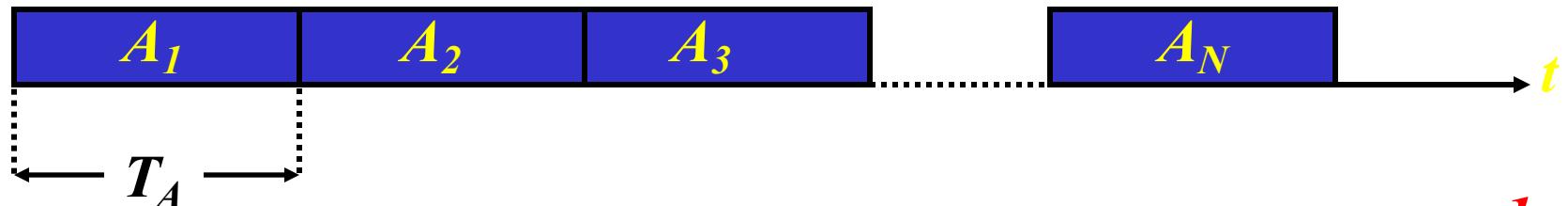
$t$



- *Latenza*: 5 fasi (clock)
- *Throughput*: a regime, dopo 5 fasi (clock), un'automobile per fase (clock)

# Principio del *Pipelining*

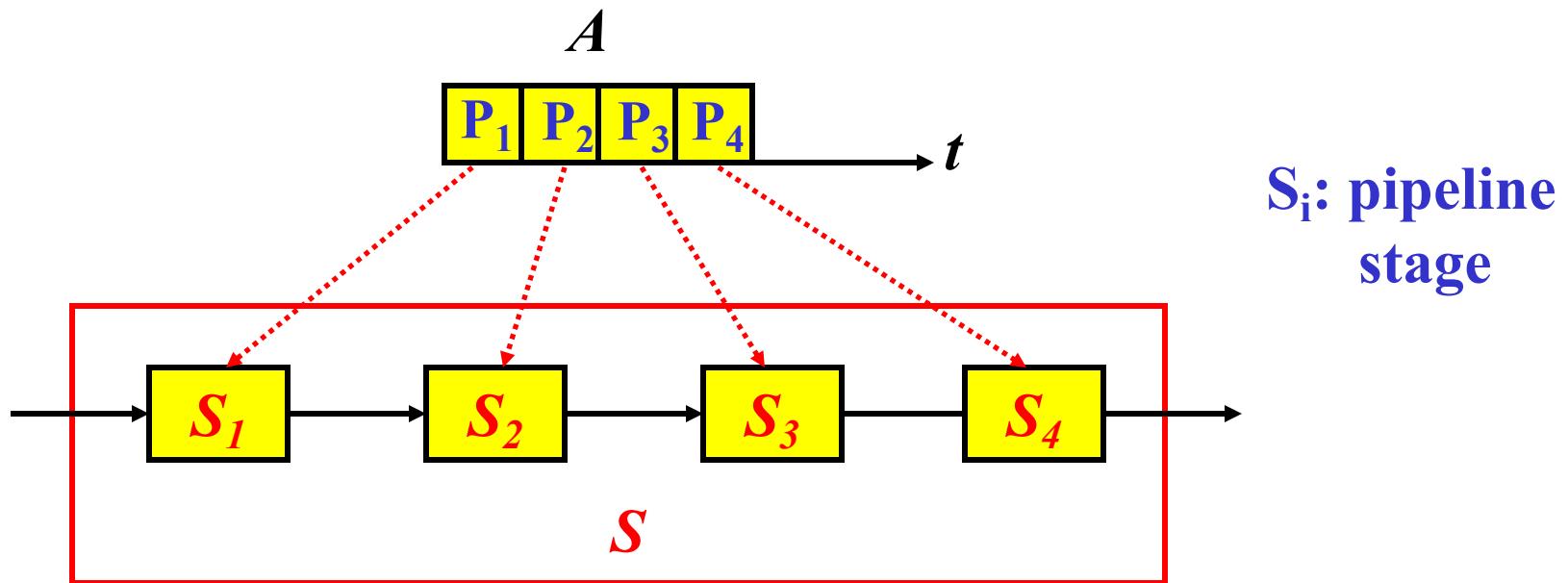
## 1) Sistema Sequenziale



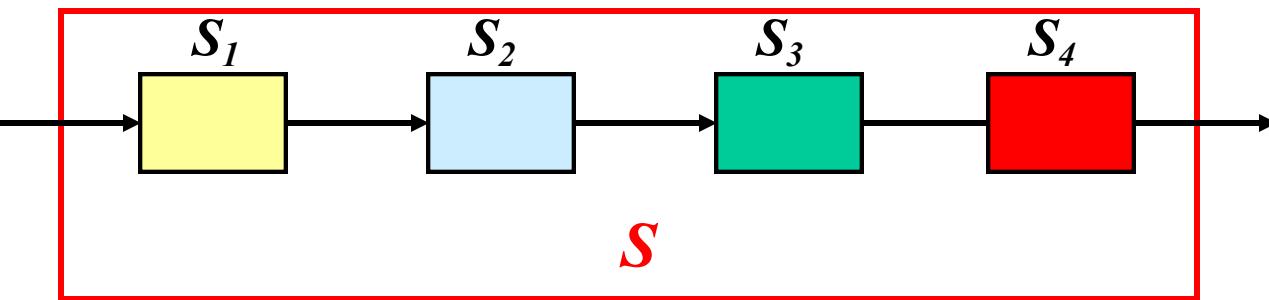
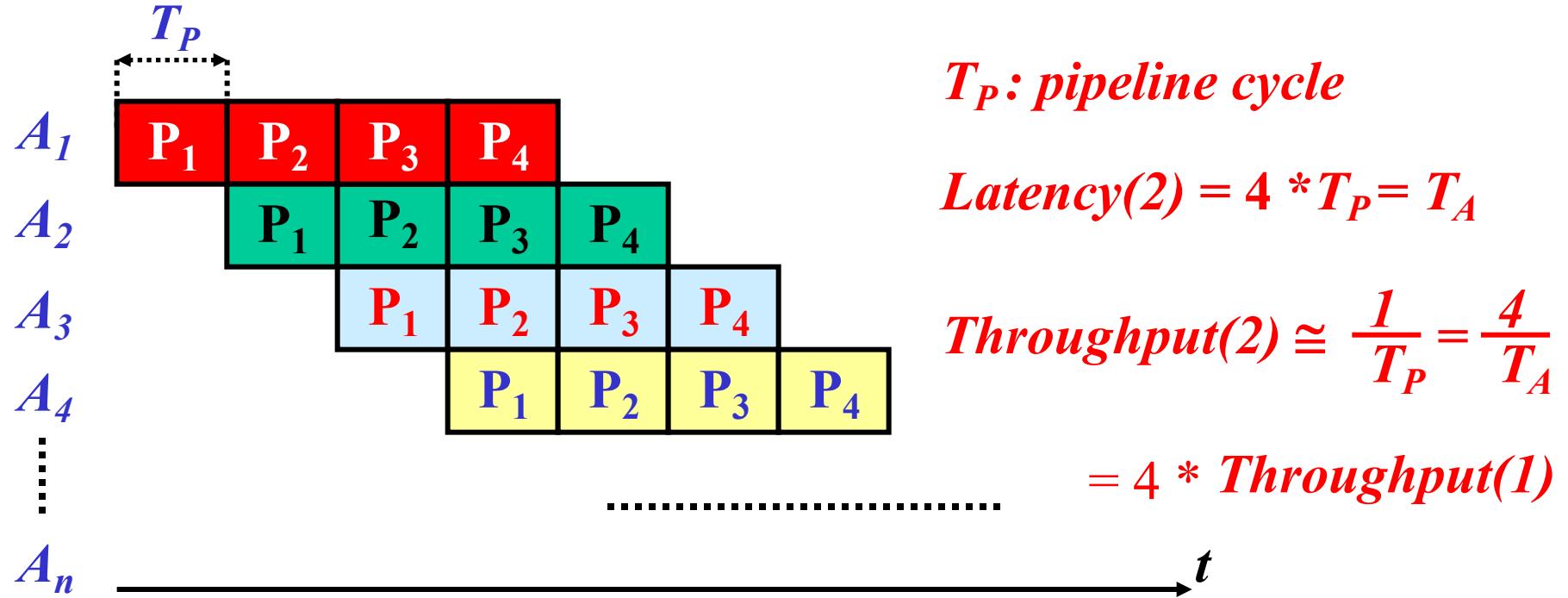
*Latency (tempo di esecuzione di una istruzione) =  $T_A$*

*Throughput =  $\frac{1}{T_A}$*

## 2) Sistema in Pipeline

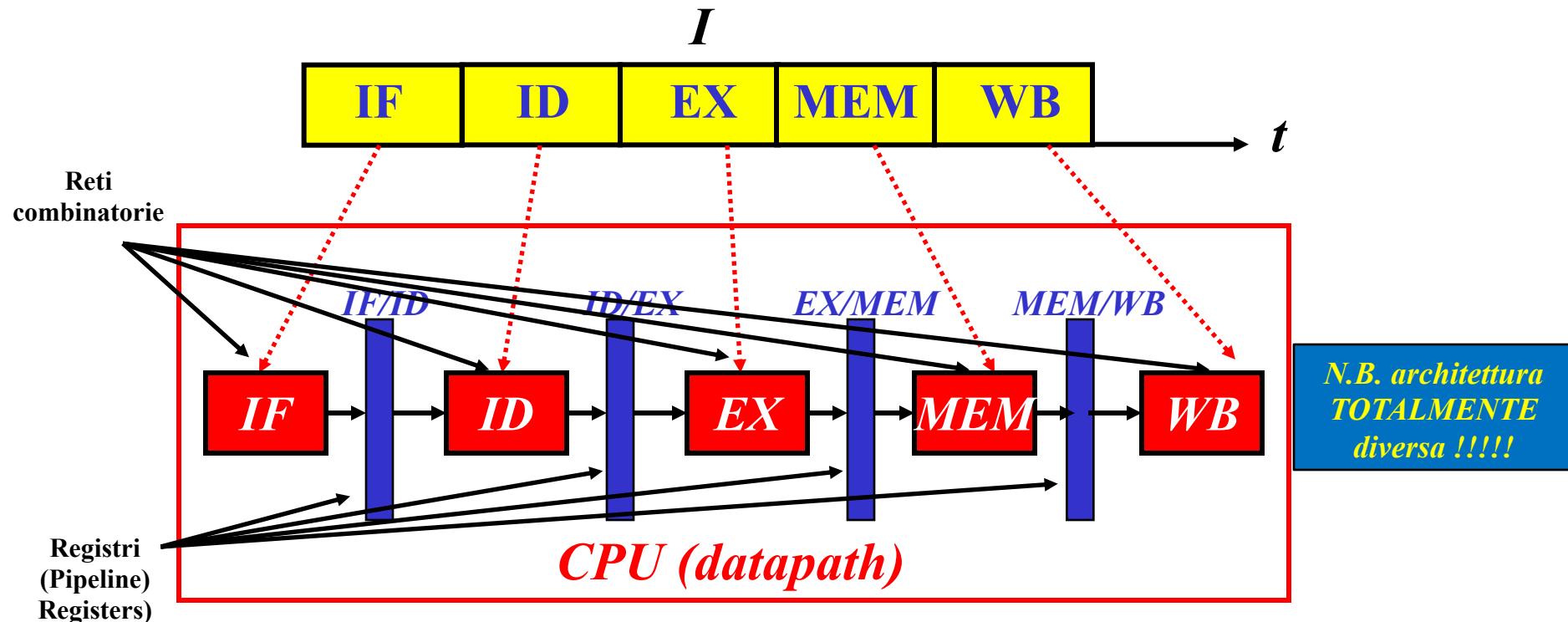


# Principio del *Pipelining*



# Pipelining in una CPU (DLX)

Attività:  $A_1, A_2, A_3 \dots A_N$  → Istruzioni:  $I_1, I_2, I_3 \dots I_N$

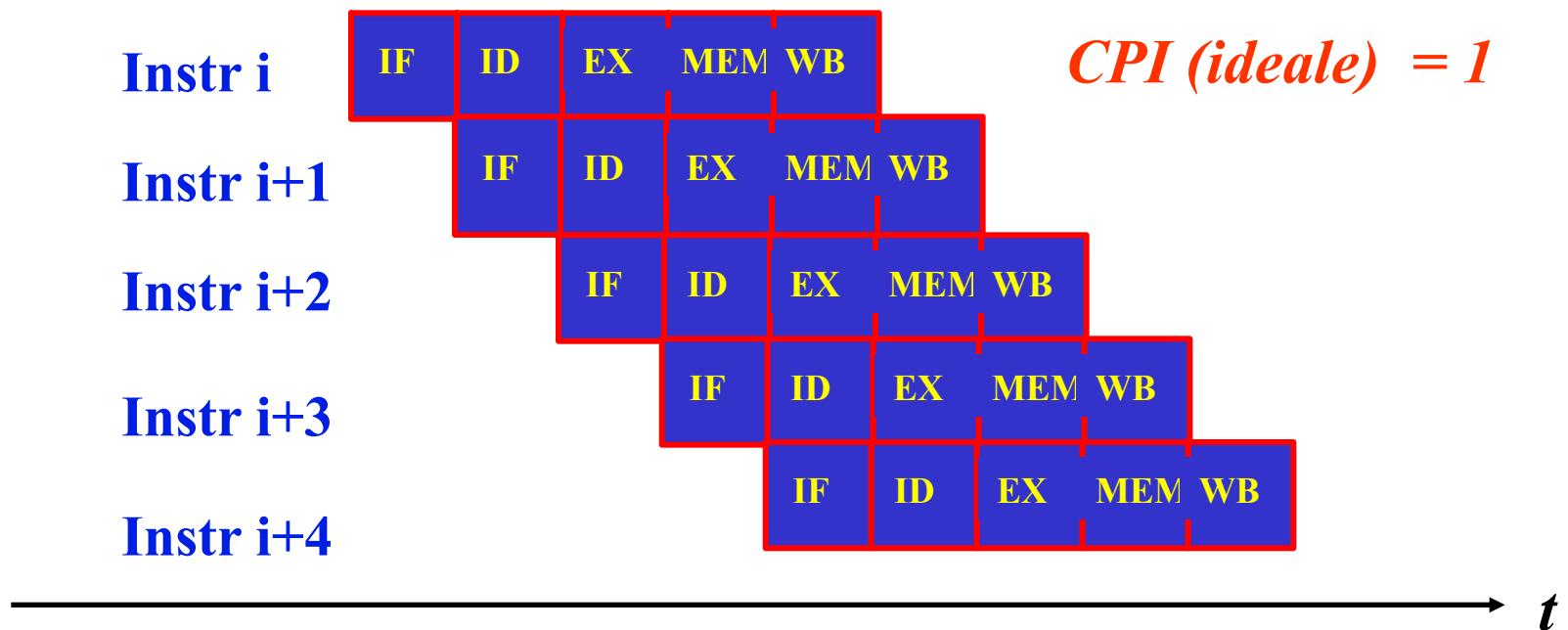


*Pipeline Cycle* → *Clock Cycle* → *Ritardo dello stadio più lento*

↓

*CPI=1 (idealemente !)*

# Pipeline del *DLX*

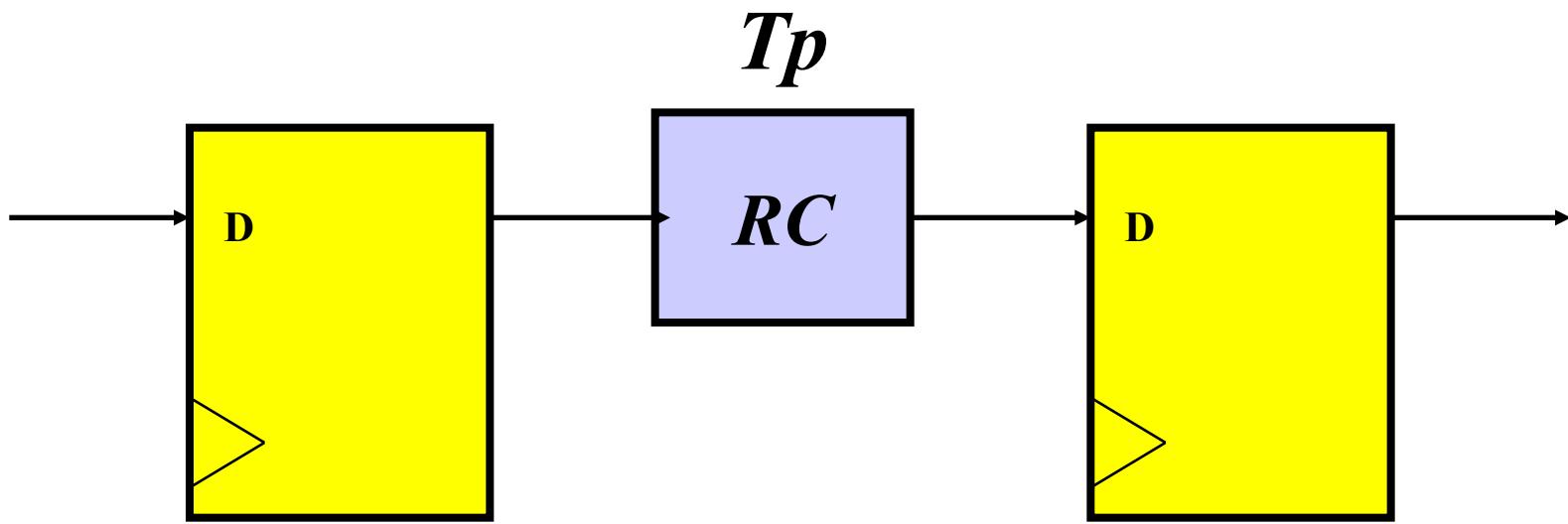


*Overhead introdotto dai Pipeline Registers:*

$$T_{clk} = T_d + T_P + T_{su}$$

Diagram illustrating the components of the clock period ( $T_{clk}$ ):

- $T_d$ : Ritardo registro a monte (Setup time of the input register)
- $T_P$ : Ritardo stadio combinatorio più lento (Combinatorial delay of the longest stage)
- $T_{su}$ : Set-up registro a valle (Setup time of the output register)



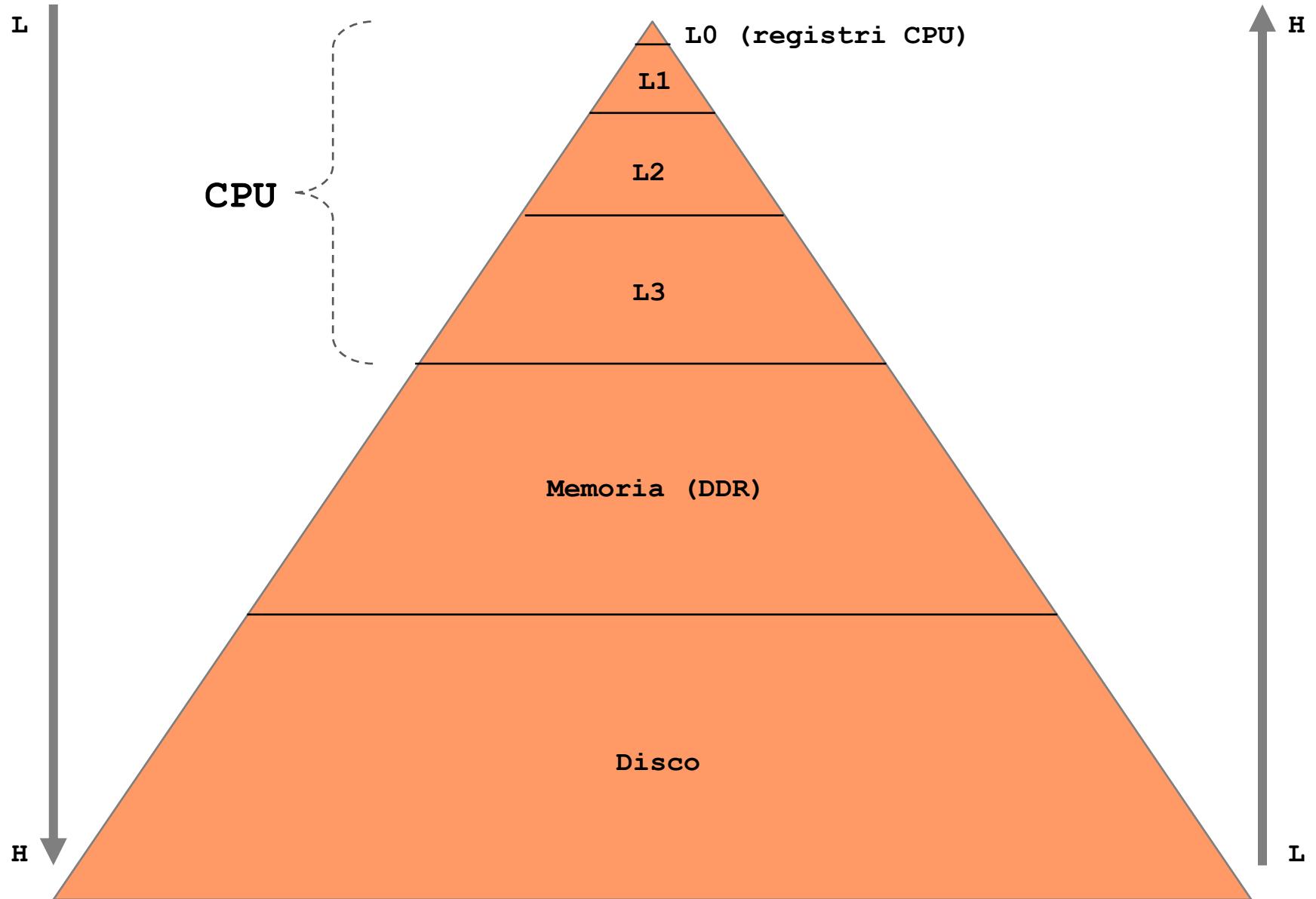
# Requisiti per l'implementazione in pipeline

- Ogni stadio deve essere attivo in ogni ciclo di clock.
- E' necessario incrementare il PC in IF (invece che in ID).
- E' necessario introdurre un ADDER ( $PC \leftarrow PC+4 - PC <- PC+1$ ) nello stadio IF.  
*uso per il calcolo di PC+4 e uno per le calcole delle istruzioni*
- Sono necessari due MDR (che chiameremo LMDR e SMDR) per gestire il caso di una LOAD seguita immediatamente da una STORE (WB-MEM sovrapposti – sovrapposizione di due dati in attesa di essere scritti, uno in memoria e l'altro nel RF).
- In ogni ciclo di clock devono poter essere eseguiti 2 accessi alla memoria (IF, MEM): Instruction Memory (IM) e Data Memory (DM)  
-> *Architettura 'Harvard'*
- Il clock della CPU è determinato dallo stadio più lento: IM, DM devono essere delle memorie *cache* (on-chip)
- I Pipeline Registers trasportano sia dati sia informazioni di controllo (l'unità di controllo è '*distribuita*' fra gli stadi della pipeline)

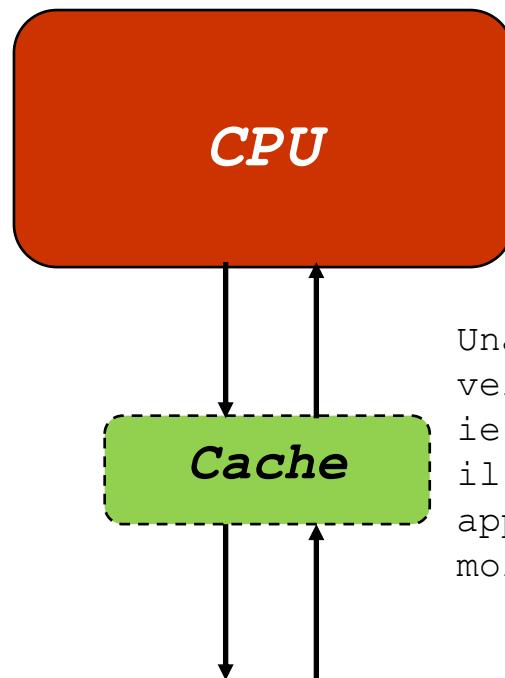
## Gerarchia della memoria

Tempo di accesso

Costo/Byte

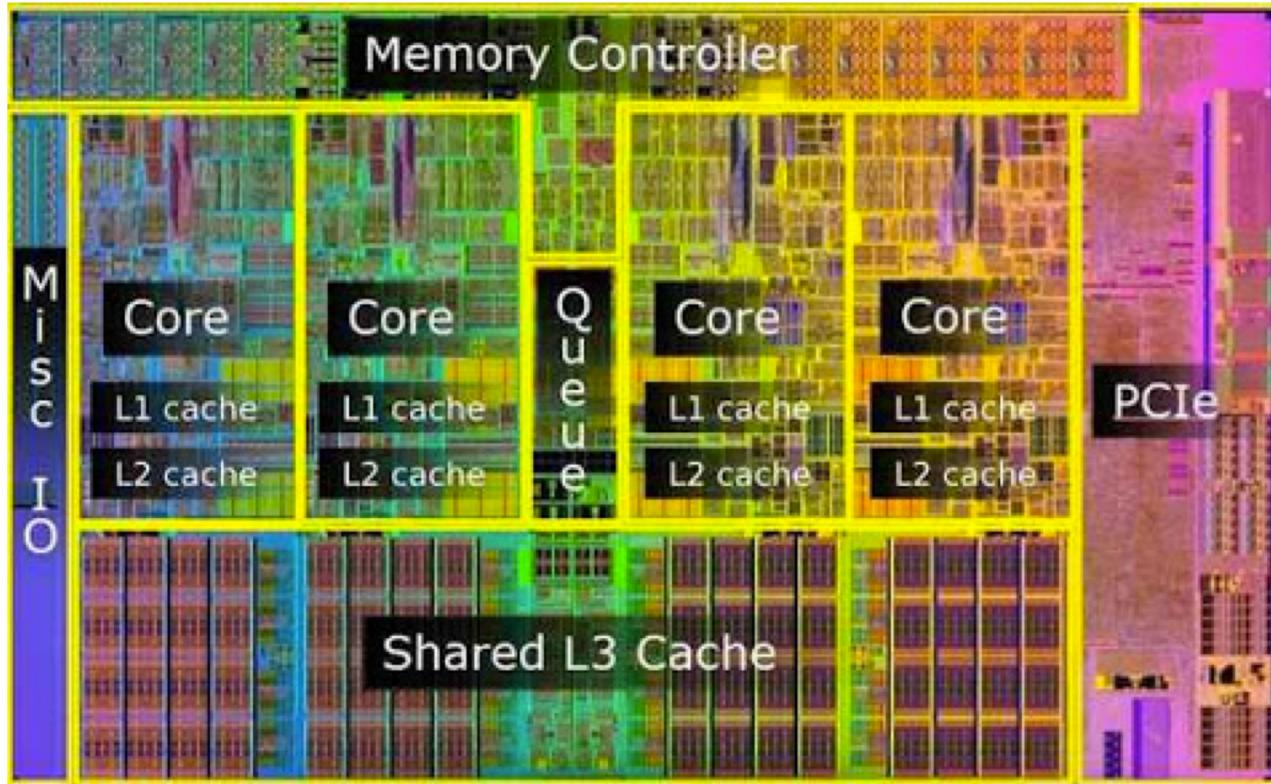


## Memorie cache



Una (o più livelli) memoria veloce ma di ridotte dimensioni, ie *cache*, in grado di sfruttare il principio di località fanno apparire la (lenta) memoria DDR molto più veloce

**Memoria (DDR)**



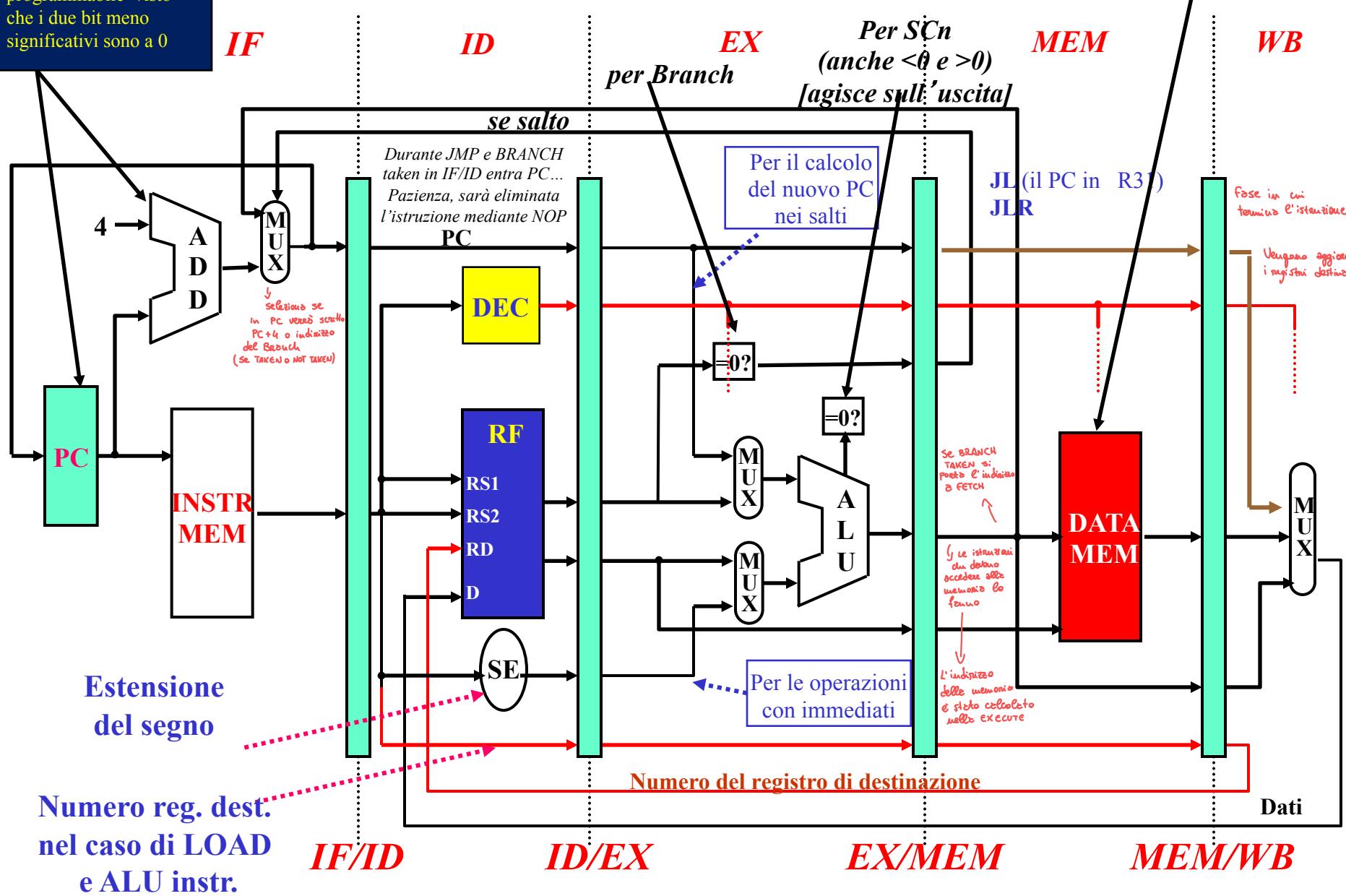
Area di silicio occupata da cache L1,L2,L3 in un Intel Core i5

Fonte: <https://thecodeartist.blogspot.com/2011/12/why-readmostly-does-not-work-as-it.html>

Contiene anche  
i circuiti di swap

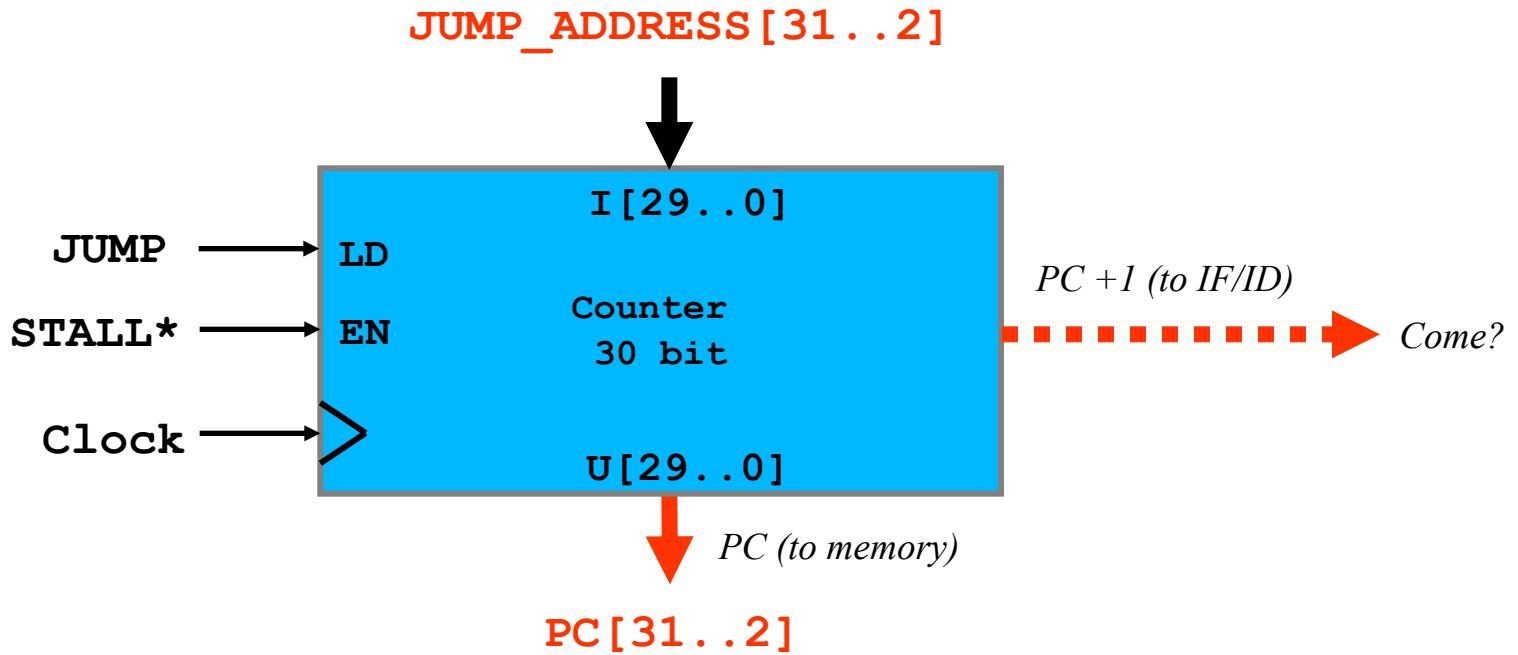
# Datapath in Pipeline del DLX

In realtà è un contatore programmabile visto che i due bit meno significativi sono a 0



## Stadio di Fetch con contatore 1/2

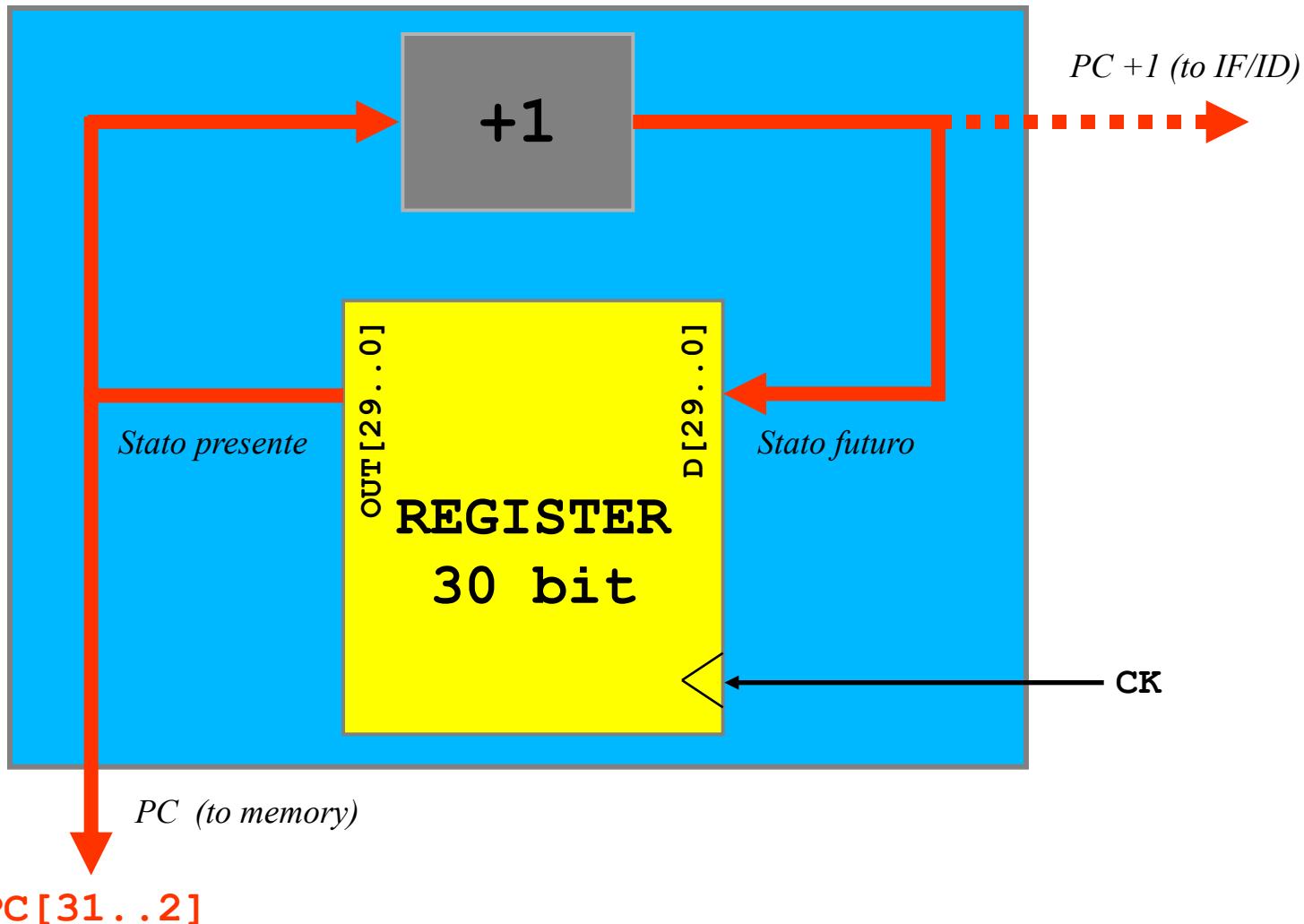
Con riferimento al primo schema del DLX *pipelined* studiato durante il corso (ma considerazioni analoghe si applicano alle altre versioni del DLX), la rete seguente consente di sostituire lo schema basato su **registro e multiplexer** con un **contatore** a 30 bit (i due bit meno significativi dell'indirizzo sono superflui perché il DLX esegue il fetch sempre a indirizzi allineati).



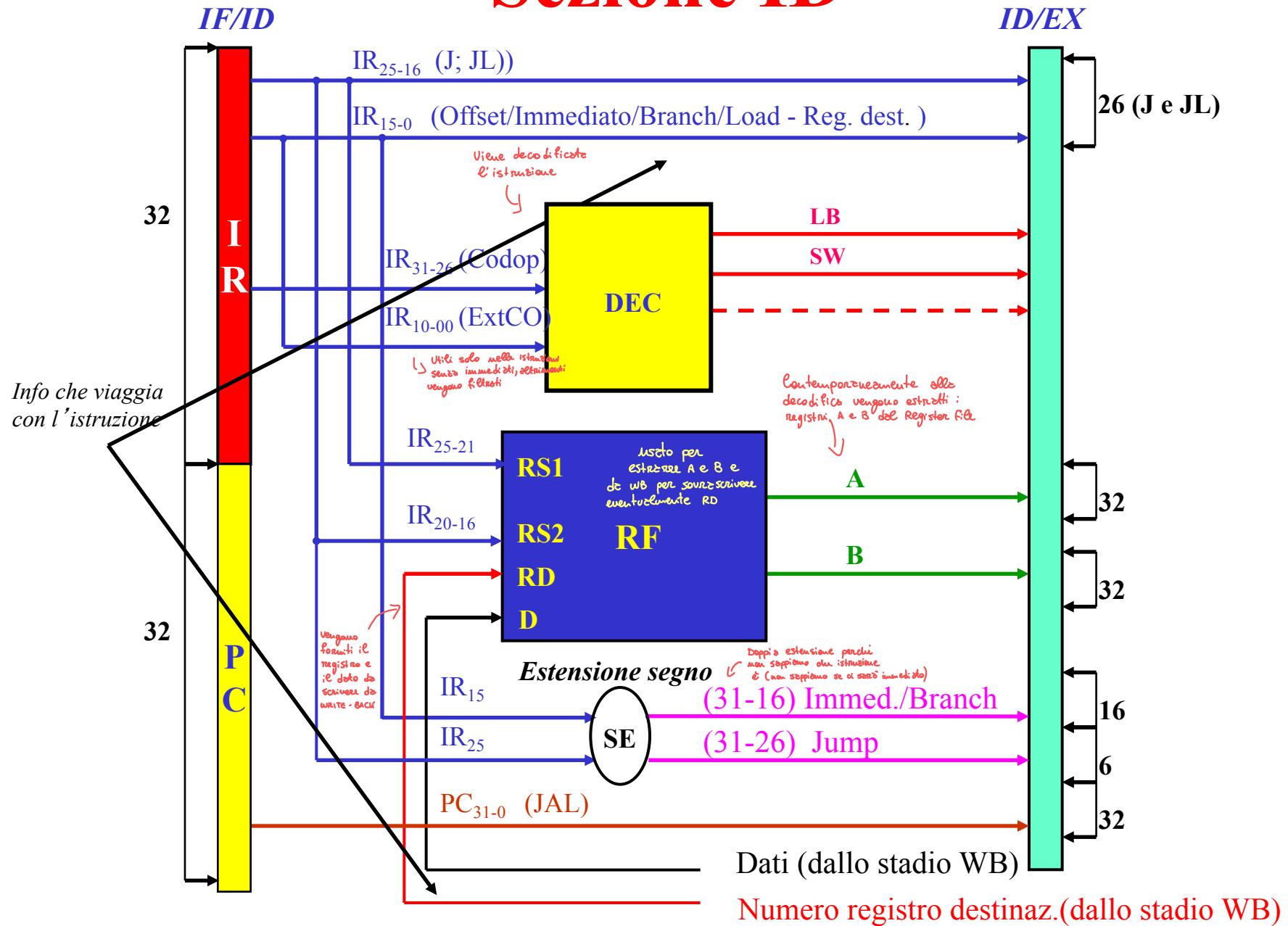
Sempre con riferimento allo stesso schema del DLX. Il segnale **JUMP** codifica se il DLX deve saltare alla destinazione specificata da **JUMP\_ADDRESS[31..2]**. Entrambi i segnali sono inviati dallo stadio **MEM**. Il segnale **STALL**, è generato dalla Unità di Controllo quando lo stadio di **IF** deve essere bloccato.

## Stadio di Fetch con contatore 2/2

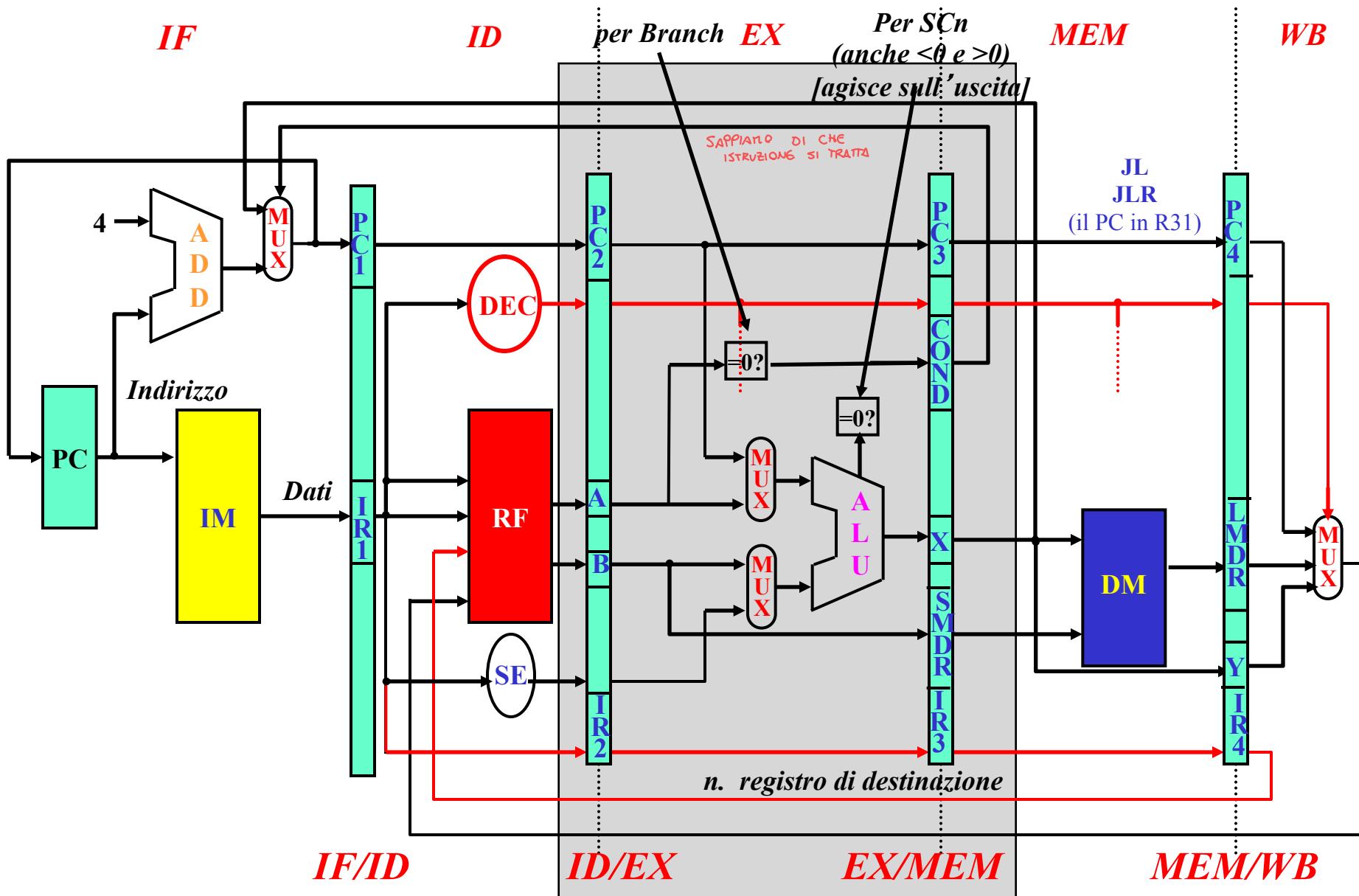
Un'osservazione: come possiamo generare  $PC + 1$  per lo stadio IF/ID (quando viene eseguito il fetch a PC è necessario fare entrare nella pipeline (stadio IF/ID)  $PC + 1$  ?



# Sezione ID



# Datapath in Pipeline del DLX



**IF/ID**

**ID/EX**

**EX/MEM**

**MEM/WB**

X: ALUOUTPUT/DMAR/BTA Y: ALUOUTPUT1

# Esecuzione in pipeline di istruzione “ALU”

*NB in questa come nelle altre istruzioni RD (RS2) è trasferita fino allo stadio WB*

|     |   |  |
|-----|---|--|
| IF  | $IR \leftarrow M[PC] ; PC \leftarrow PC + 4 ; PC1 \leftarrow PC + 4$  |  |
| ID  | $A \leftarrow RS1; B \leftarrow RS2; PC2 \leftarrow PC1; IR2 \leftarrow IR1$<br><del>ID/EX &lt;- Decodifica istruzione;</del>                                     | <i>N.B. al passare degli stadi IR perde i bit che non servono più in tutte le istruzioni. Da uno stadio al successivo vengono mantenuti i bit che servono qualunque sia l'istruzione</i> |
| EX  | OPERAZIONE TRA DUE REGISTRI O TRA REGISTRO E IMMEDIATO<br>$X \leftarrow A \text{ op } B$<br>oppure<br>$X \leftarrow A \text{ op } (IR2_{15})^{16} \# IR2_{15..0}$ | [PC3 <- PC2]<br>[IR3 <- IR2]<br><i>EXTENSIONS GIA' CALCOLATA CON ID</i>  |
| MEM | $Y \leftarrow X$ (“parcheggio” in attesa di WB)<br><i>NON DEVE ESSERE NULLA</i>   | [IR4 <- IR3] [PC4 <- PC3]  |
| WB  | $RD \leftarrow Y$   | <i>VIENE MODIFICATO IL REGISTRO DESTINAZIONE</i>   |

*X : “ALUOUTPUT” (in EX/MEM), Y: “ALUOUTPUT1”*

# Esecuzione in pipeline di istruzione “MEM”

|     |   |
|-----|---|
| IF  | $IR \leftarrow M[PC] ; PC \leftarrow PC + 4 , PC1 \leftarrow PC + 4$  |
| ID  | $A \leftarrow RS1; B \leftarrow RS2; PC2 \leftarrow PC1; IR2 \leftarrow IR1$<br><del>ID/EX &lt;- Decodifica istruzione;</del>   |
| EX  | $MAR \leftarrow A \text{ op } (IR2_{15})^{16} \# IR2_{15..0}$<br>$SMDR \leftarrow B$ <div style="position: absolute; left: -100px; top: 50px; color: red; font-size: small;">           La ALU viene utilizzata per calcolare l'indirizzo del registro sorgente         </div> <div style="position: absolute; right: 0; top: 0; color: blue; font-size: small;"> <math>[IR3 \leftarrow IR2]</math><br/> <math>[PC3 \leftarrow PC2]</math> </div> |
| MEM | $LMDR \leftarrow M[MAR]$ ( <i>LOAD</i> )<br>oppure<br>$M[MAR] \leftarrow SMDR$ ( <i>STORE</i> )   |
| WB  | $RD \leftarrow LMDR$ ( <i>LOAD</i> ) [ext. Segno]   |

La decodifica attraversa tutti gli stadi



Questo stadio potrebbe rallentare l'esecuzione delle pipeline se il dato non è nelle cache. Vengono fermate le fasi precedenti sia quelle successive.

# Esecuzione in pipeline di istruzione “BRANCH”

|     |   |
|-----|---|
| IF  | $IR \leftarrow M[PC] ; PC \leftarrow PC + 4 ; PC1 \leftarrow PC + 4$  |
| ID  | $A \leftarrow RS1; B \leftarrow RS2; PC2 \leftarrow PC1; IR2 \leftarrow IR1$<br><b>ID/EX</b> <- Decodifica istruzione;  |
| EX  | <p>Se il BRANCH è NOT TAKEN si fa il FETCH a <math>PC + 4</math></p> <p>Altroimenti bisogna calcolare l'indirizzo di destinazione del salto (PC relativa)</p> $X \leftarrow PC2 \text{ op } (IR_{15})^{16} \# IR_{15..0}$ $\text{Cond} \leftarrow A \text{ op } 0$ <span style="float: right;"><math>[PC3 \leftarrow PC2]</math><br/><math>[IR3 \leftarrow IR2]</math></span> |
| MEM | <p>La retroazione è fatta da MEM per evitare dei puntatori dovrà ai calcoli</p> $\text{if } (\text{Cond}) PC \leftarrow X$ <span style="float: right;"><math>[PC4 \leftarrow PC3]</math><br/><math>[IR4 \leftarrow IR3]</math></span>   |
| WB  | <p>Non viene fatto nulla</p> $(\text{NOP})$   |

La decod.ifica attraversa tutti gli stadi

$X$  : “BTA (BRANCH TARGET ADDRESS)”

*Il test avviene sul valore del registro*

# Esecuzione in pipeline di un' istruzione “JR”

|            |   |
|------------|---|
| <b>IF</b>  | $IR \leftarrow M[PC]$ ; $PC \leftarrow PC + 4$ ; $PC1 \leftarrow PC + 4$  |
| <b>ID</b>  | $A \leftarrow RS1$ ; $B \leftarrow RS2$ ; $PC2 \leftarrow PC1$ ; $IR2 \leftarrow IR1$<br><b>ID/EX</b> <- Decodifica istruzione;;          |
| <b>EX</b>  | Si salta a un indirizzo assoluto<br>(non PC relative)<br>$X \leftarrow A$ [IR3 <- IR2]<br>[PC3 <- PC2]                                    |
| <b>MEM</b> | Venne fatta la retroazione e viene inviato alle fasi di FETCH la destinazione del salto<br>$PC \leftarrow X$ [IR4 <- IR3]<br>[PC4 <- PC3] |
| <b>WB</b>  | (NOP)   |

La decod.ifica attraversa tutti gli stati



Come sarebbe la sequenza degli stati per una J ?

Il salto è  
↓ PC relative  
(dovrò sommare PC + Imm (26 bit → 32bit))

# Esecuzione in pipeline di istruzione “JL o JLR”

*Si gestiscono in modo analogo alle JUMP*

|        |  |
|--------|--|
| I<br>F | $IR \leftarrow M[PC] ; PC \leftarrow PC + 4 ; PC1 \leftarrow PC + 4$   |
| ID     | $A \leftarrow RS1; B \leftarrow RS2; PC2 \leftarrow P1; IR2 \leftarrow IR1$<br><del>ID/EX &lt;- Decodifica istruzione;</del>                                       |
| EX     | $PC3 \leftarrow PC2$ [IR3 <- IR2]<br>$X \leftarrow A$ (Se JLR) $X \leftarrow PC2 + (IR_{25})^6 \# IR_{25..0}$ (Se JL)  |
| MEM    | Dobbiamo salvare<br>PC in R31 quindi<br>ce lo posiziono dietro<br>$\hookrightarrow PC \leftarrow X ; PC4 \leftarrow PC3$ [IR4 <- IR3]                              |
| WB     | E' bene che in ogni studio<br>ci sia una sola scrittura<br>su Register File<br>$R31 \leftarrow PC4$<br><i>Evidenziati perché<br/>in questo caso<br/>utilizzati</i> |

Decod. in  
tutti gli stadi

NB: La scrittura in R31 NON può essere anticipata perché  
potrebbe sovrapporsi ad altra scrittura di registro

# Qual sarebbe la sequenza nel caso di SCN (ex SLT R1,R2,R3) ?

|        |   |
|--------|---|
| I<br>E | $IR \leftarrow M[PC]$ ; $PC \leftarrow PC + 4$ ; $PC1 \leftarrow PC + 4$  |
| ID     | $A \leftarrow RS1$ ; $B \leftarrow RS2$ ; $PC2 \leftarrow P1$ ; $IR2 \leftarrow IR1$<br><b>ID/EX &lt;- Decodifica istruzione;</b> |
| EX     | $X \leftarrow R2 \text{ op } R3$<br>?   |
| MEM    | Non viene fatto nulla<br>?  |
| WB     | Si scrive l'esito dell'operazione nel registro destinazione<br>?  |

# Alee nelle Pipeline

Si verifica una situazione di “*Alea*” (“*Hazard*”) quando in un determinato ciclo di clock un’ istruzione presente in uno stadio della pipeline non può essere eseguita in quel clock.

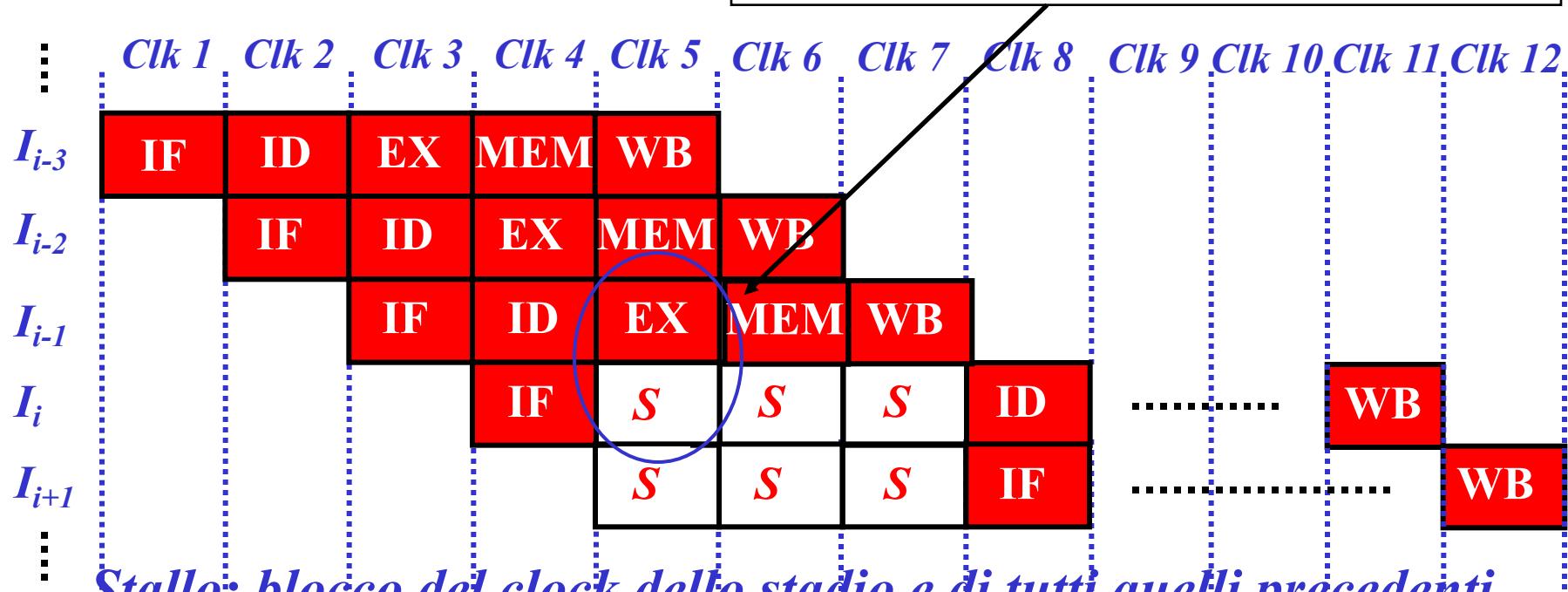
- **Alee Strutturali** - Una risorsa è condivisa fra due stadi della pipeline: le istruzioni che si trovano correntemente in tali stadi non possono essere eseguite simultaneamente.
- **Alee di Dato** – Sono dovute a *dipendenze* fra le istruzioni. Ad esempio una istruzione che legge un registro scritto da un’ istruzione precedente (**RAW**).  
(READ AFTER WRITE)
- **Alee di Controllo** – Le istruzioni che seguono un branch dipendono dal risultato del branch (*taken/not taken*).



L’ istruzione che non può essere eseguita viene bloccata (“*stallo della pipeline*”), insieme a tutte quelle che la seguono, mentre le istruzioni che la precedono avanzano normalmente (così da rimuovere la causa dell’alea).

# Alee e Stalli

*Effetto – ad esempio – di una alea di dato: se l’istruzione  $I_i$  necessita di un dato prodotto dalla istruzione  $I_{i-1}$  deve aspettare fino al WB della  $I_{i-1}$*



*Stallo: blocco del clock dello stadio e di tutti quelli precedenti  
e propagazione progressiva agli stadi successivi*

$$T_5 = 8 * CLK = (5 + 3) * CLK$$



$$T_5 = 5 * (1 + \frac{3}{5}) * CLK$$



*CPI ideale*

*Stalli per istruzione*

$$T_N = N * 1 * CLK$$

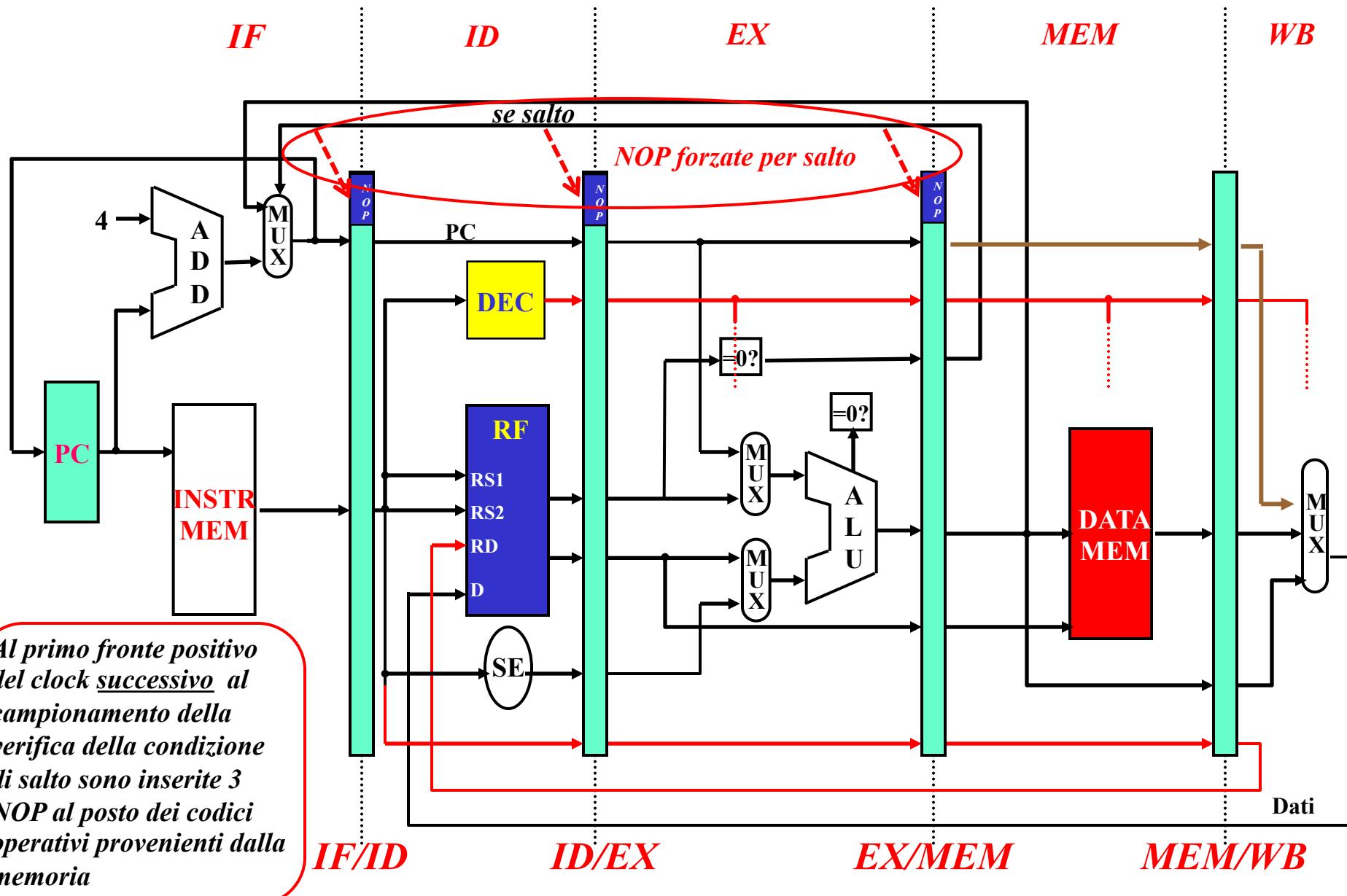


$$T_N = N * (1 + S) * CLK$$



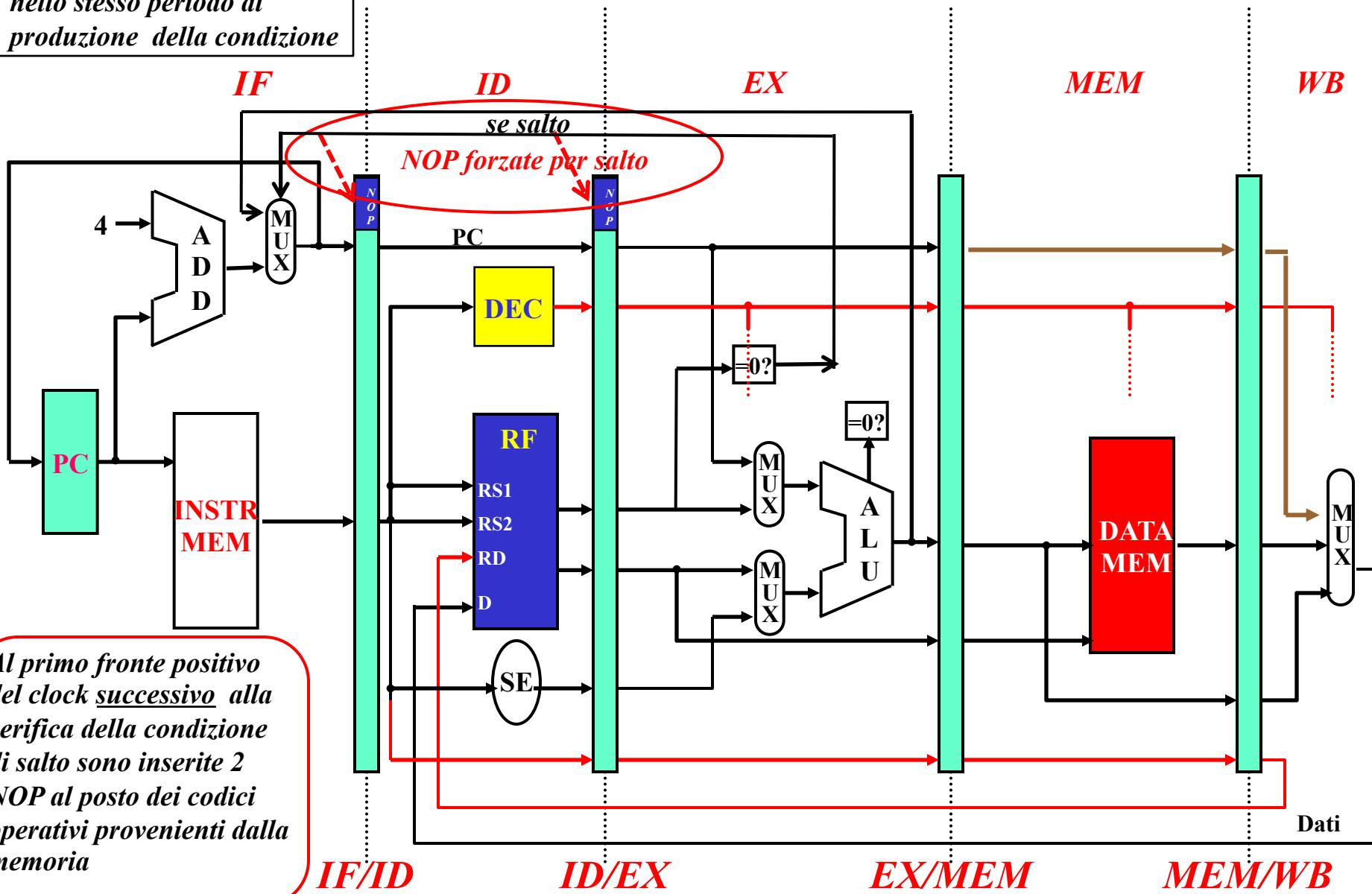
*CPI effettivo*

# Stalli nel salto (1/3)



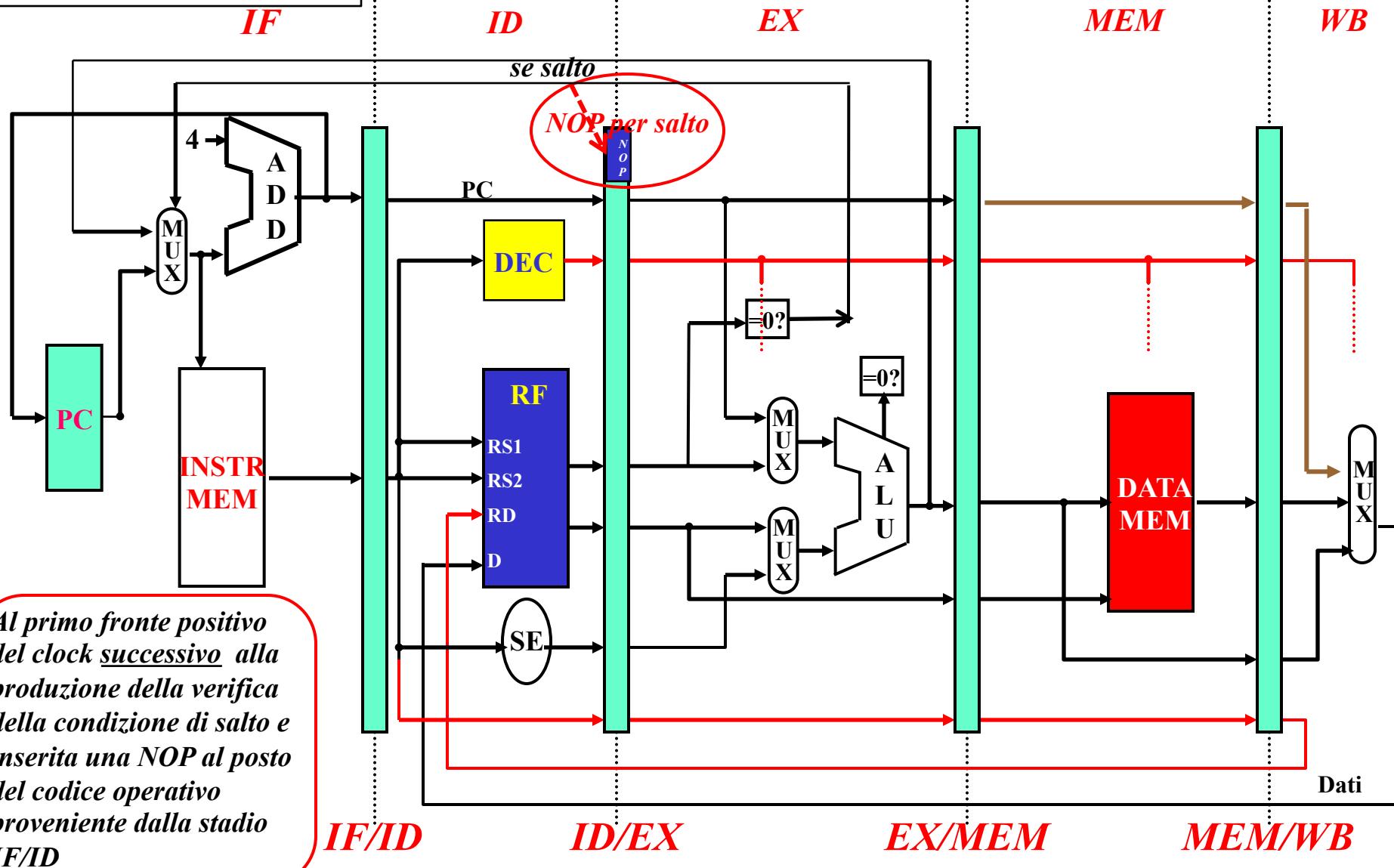
*NB In questo caso la condizione di salto e il nuovo PC sono presentati al MUX nello stesso periodo di produzione della condizione*

## Stalli nel salto (2/3)



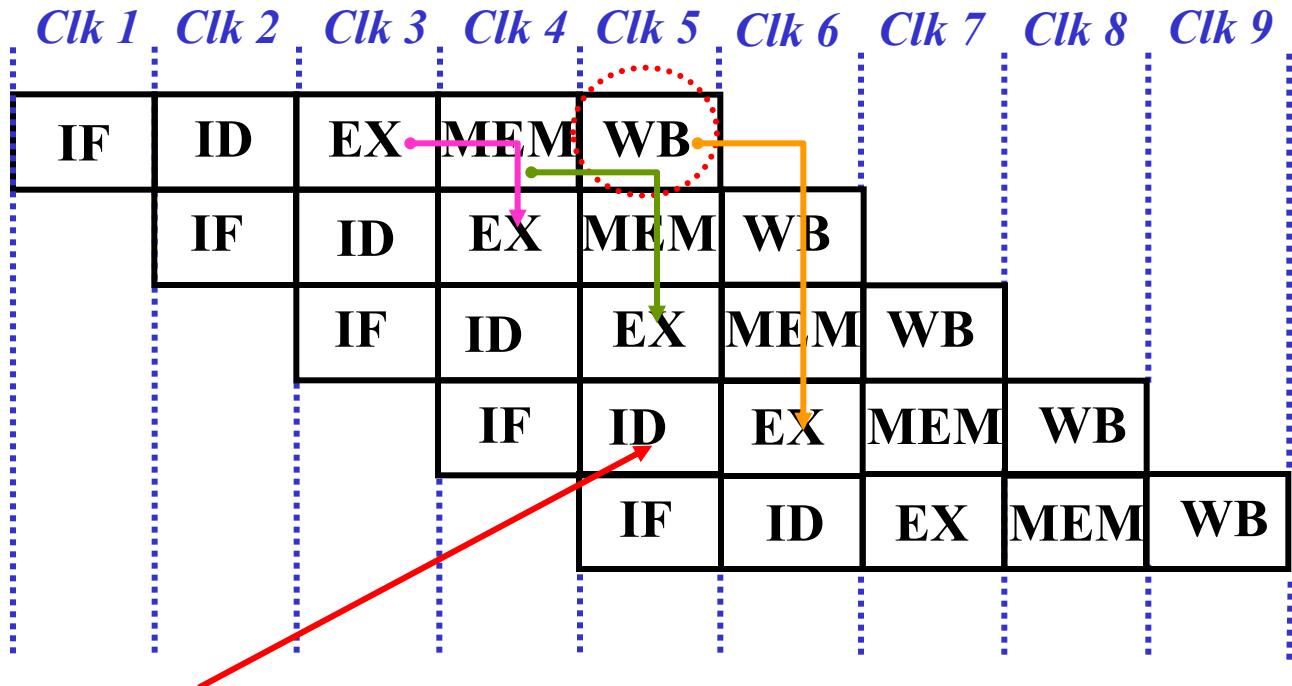
*NB In questo caso la condizione di salto e il nuovo PC agiscono sul MUX nello stesso periodo di produzione della condizione*

# Stalli nel salto (3/3)



# Forwarding

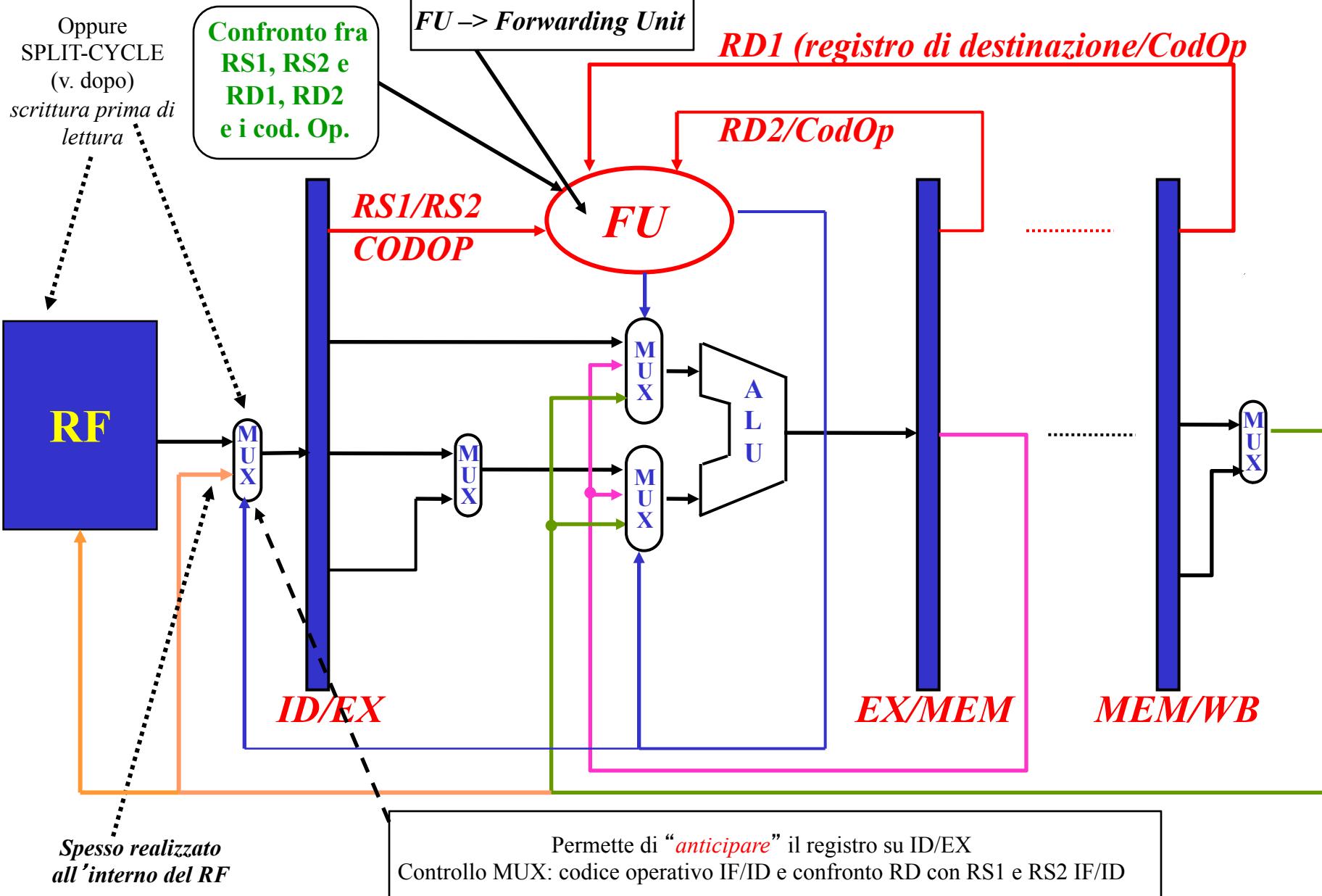
ADD R3, R1, R4  
 SUB R7, R3, R5 *alea*  
 OR R1, R3, R5 *alea*  
 LW R6, 100 (R3) *alea*  
 AND R9, R5, R3 *no alea*



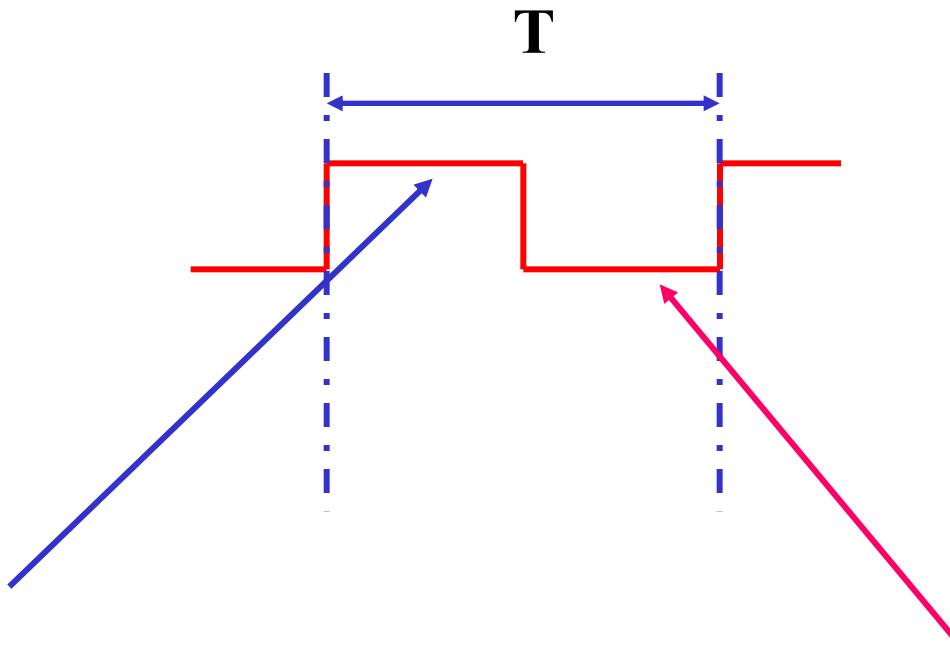
Anche qui il dato non è ancora in RF per essere estratto in ID !

**Il *forwarding* consente di eliminare quasi tutte le alee di tipo RAW della pipeline del DLX senza stallare la pipeline.**  
**(NB: nel DLX si alterano i registri solo in WB)**

# Implementazione del Forwarding



# Split-cycle



In questo  
semiperiodo si  
*scrive* il registro

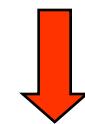
In questo  
semiperiodo si  
*legge* il registro

# Alea di dato dovuta alle istruzioni di LOAD

|              |    |    |    |     |    |
|--------------|----|----|----|-----|----|
| LW R1,32(R6) | IF | ID | EX | MEM | WB |
| ADD R4,R1,R7 | IF | ID | EX | MEM |    |
| SUB R5,R1,R8 |    | IF | ID | EX  |    |
| AND R6,R1,R7 |    |    | IF | ID  |    |

N.B. il dato richiesto dalla ADD è presente solo *alla fine* di MEM. L'alea non può essere eliminata con il forwarding (a meno di non aprire una ulteriore di ingresso ai mux della ALU dalla memoria – ritardi !)

*Di fatto non viene generato il clock.  
Il blocco di un clock si propaga  
lungo la pipeline uno stadio alla  
volta.*



E' necessario stallare la  
pipeline

Dalla fine di  
questo stadio in  
poi normale  
forwarding

|              |    |    |    |     |     |
|--------------|----|----|----|-----|-----|
| LW R1,32(R6) | IF | ID | EX | MEM | WB  |
| ADD R4,R1,R7 | IF | ID | S  | EX  | MEM |
| SUB R5,R1,R8 |    | IF | S  | ID  | EX  |
| AND R6,R1,R7 |    |    | S  | IF  | ID  |

# Delayed load

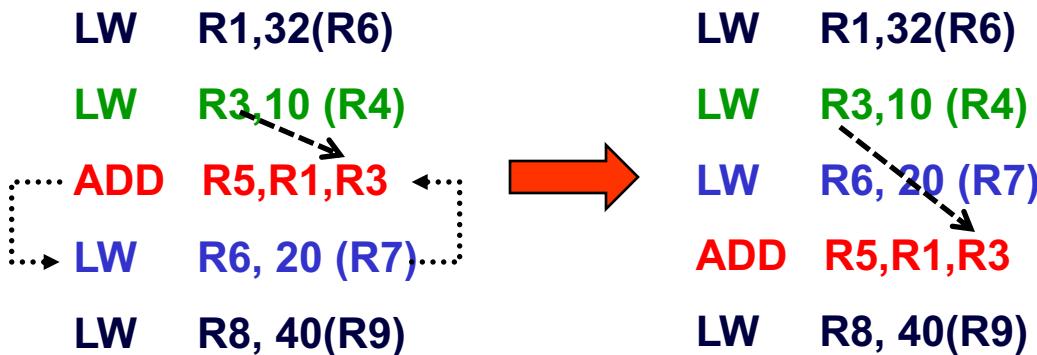
In diverse CPU RISC l' alea associata alla LOAD non è gestita in HW stallando la pipeline ma è gestita via SW dal compilatore (*delayed load*):

Istruzione LOAD

delay slot

Istruzione Successiva

Il compilatore cerca  
di riempire il delay-slot  
con un' istruzione “utile”  
(caso peggiore: NOP).

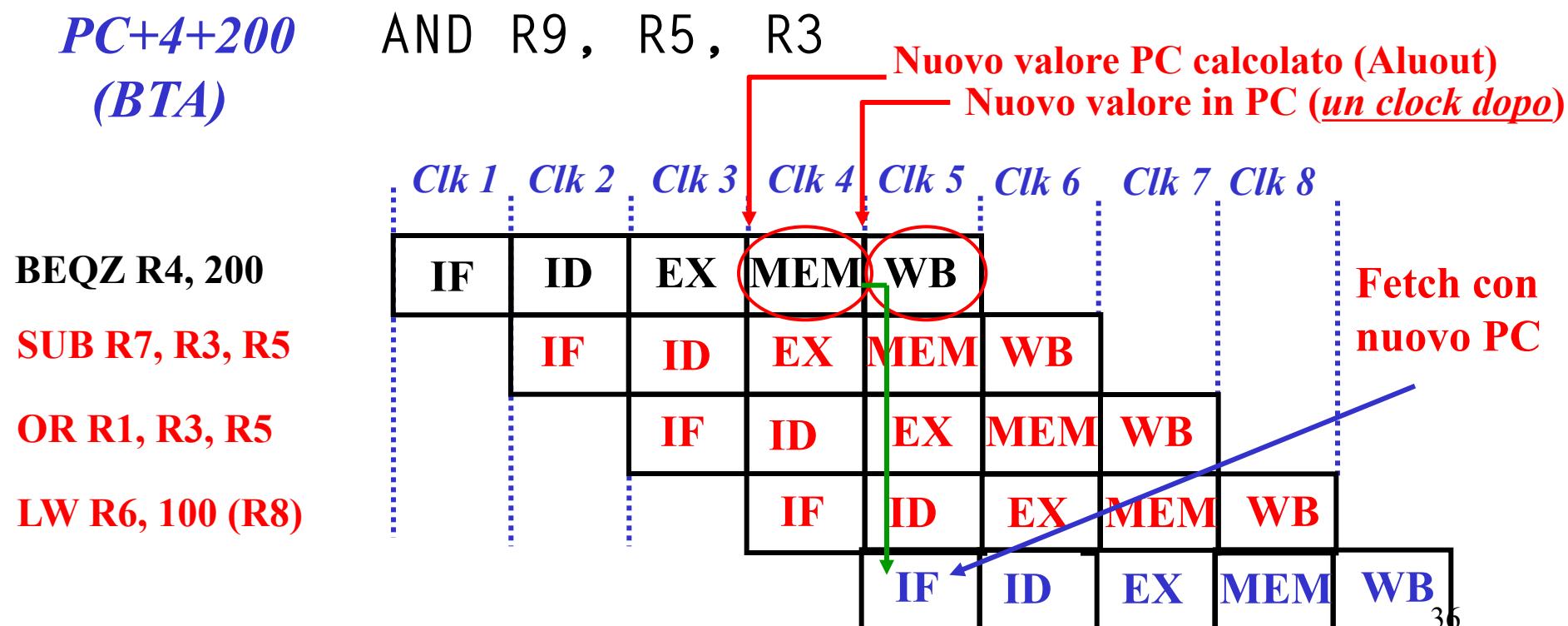


# Alee di Controllo

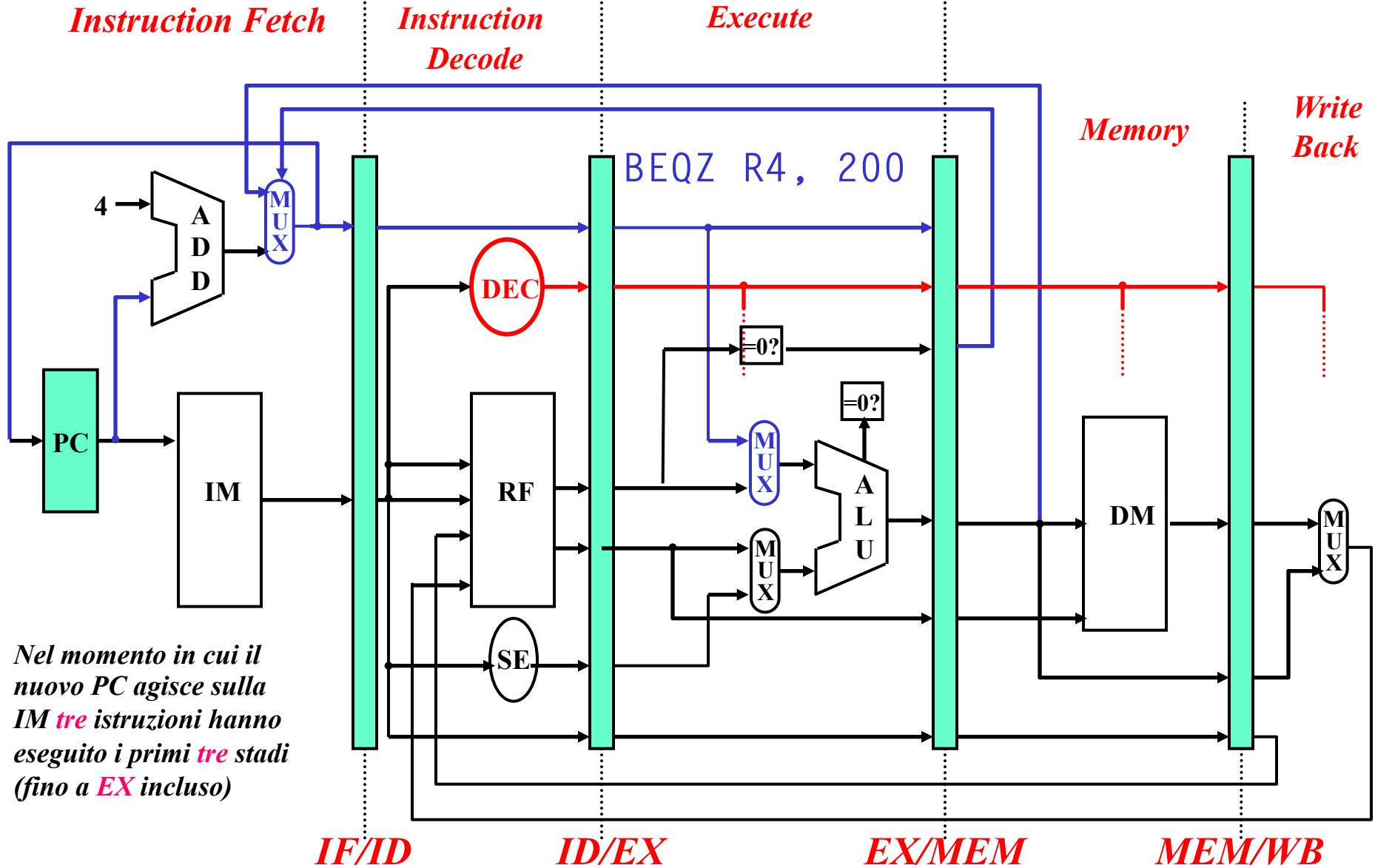
*Next Instruction Address*

|                                 |                  |  |  |
|---------------------------------|------------------|--|--|
| <i>PC</i>                       | BEQZ R4 , 200    |  |  |
| <i>PC+4</i>                     | SUB R7 , R3 , R5 |  |  |
| <i>PC+8</i>                     | OR R1 , R3 , R5  |  |  |
| <i>PC+12</i>                    | LW R6 , 100 (R8) |  |  |
|                                 |                  |  |  |
| <i>PC+4+200</i><br><i>(BTA)</i> | AND R9 , R5 , R3 |  |  |

R4 = 0 : *Branch Target Address*  
(taken)  
R4 ≠ 0 : *PC+4*  
(not taken)

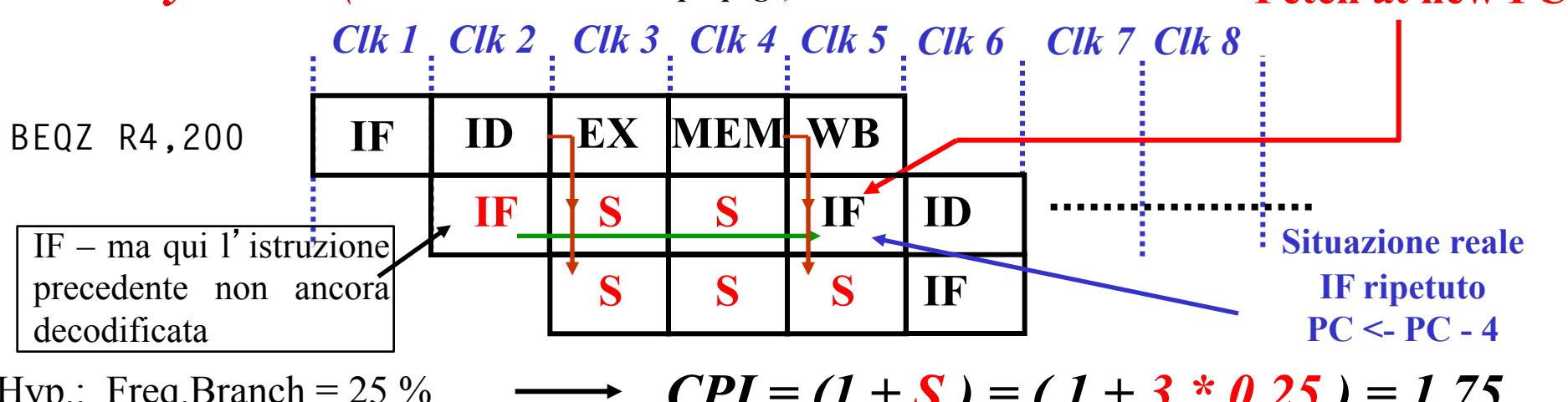


# Datapath in Pipeline del DLX (caso 1/3) - (Branch o JMP)

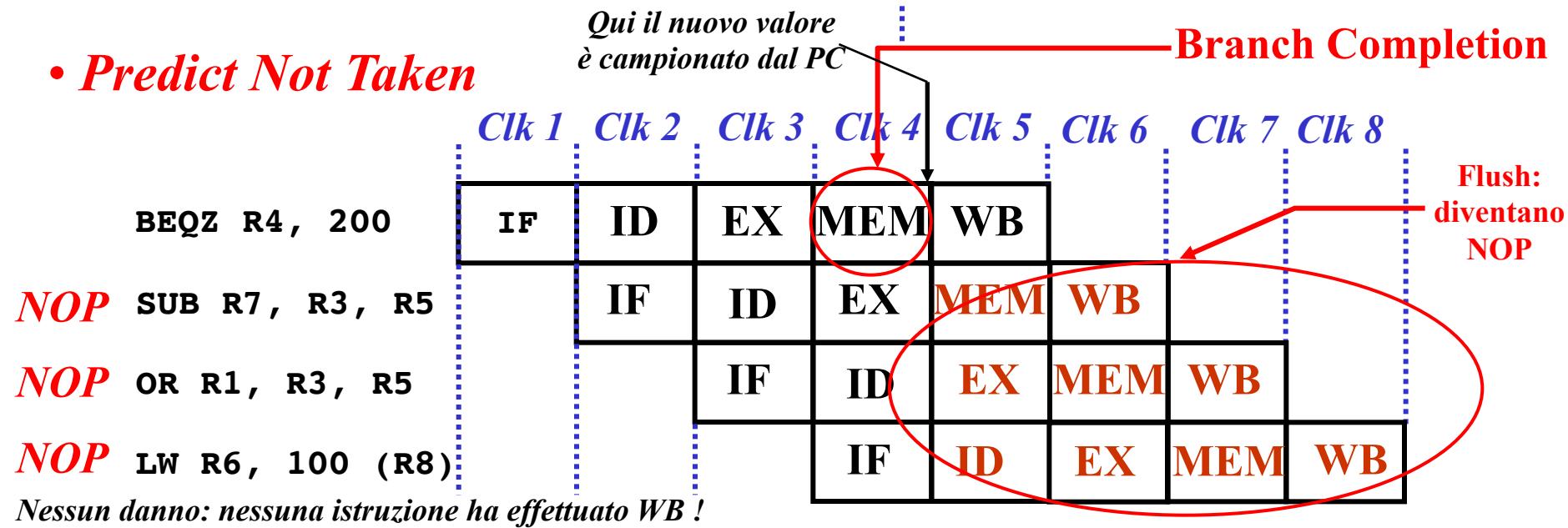


# Gestione delle Alee di Controllo

- *Always Stall* (*blocco di tre clock che si propaga*)



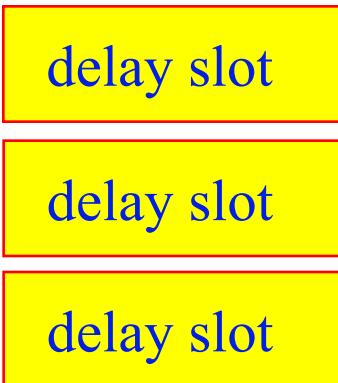
- *Predict Not Taken*



# Delayed branch

Similmente al caso della LOAD, in diverse CPU di tipo RISC l' alea associata alle istruzioni di BRANCH è gestita via SW dal compilatore (*delayed branch*):

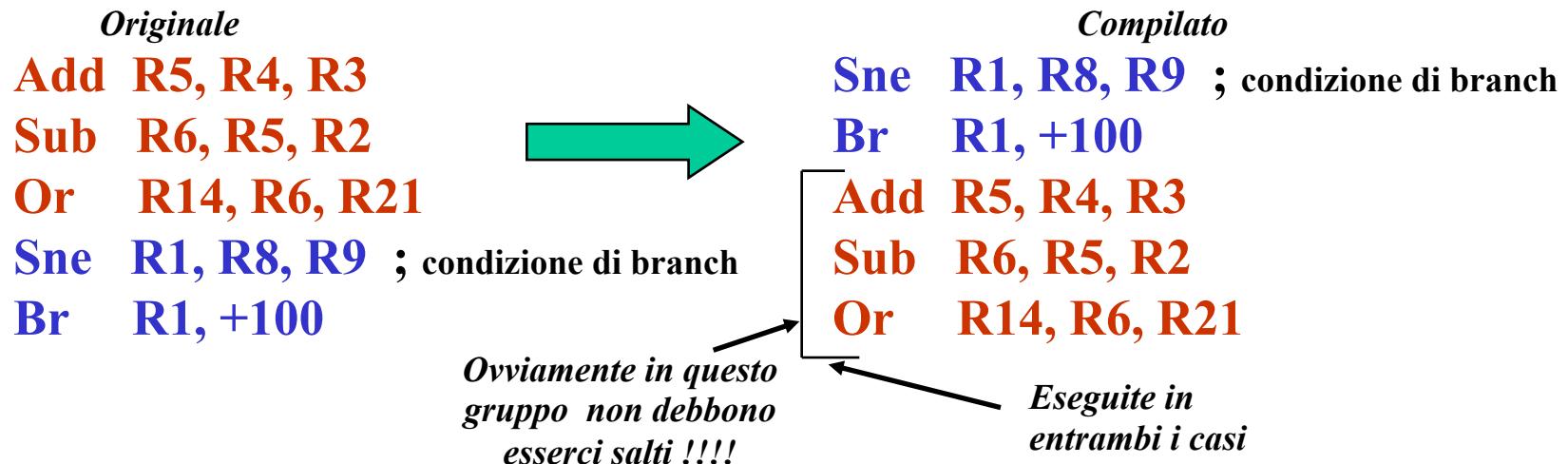
Istruzione BRANCH



Il compilatore cerca di riempire i delay-slot con istruzioni “utili” (caso peggiore: NOP).

Istruzione Successiva

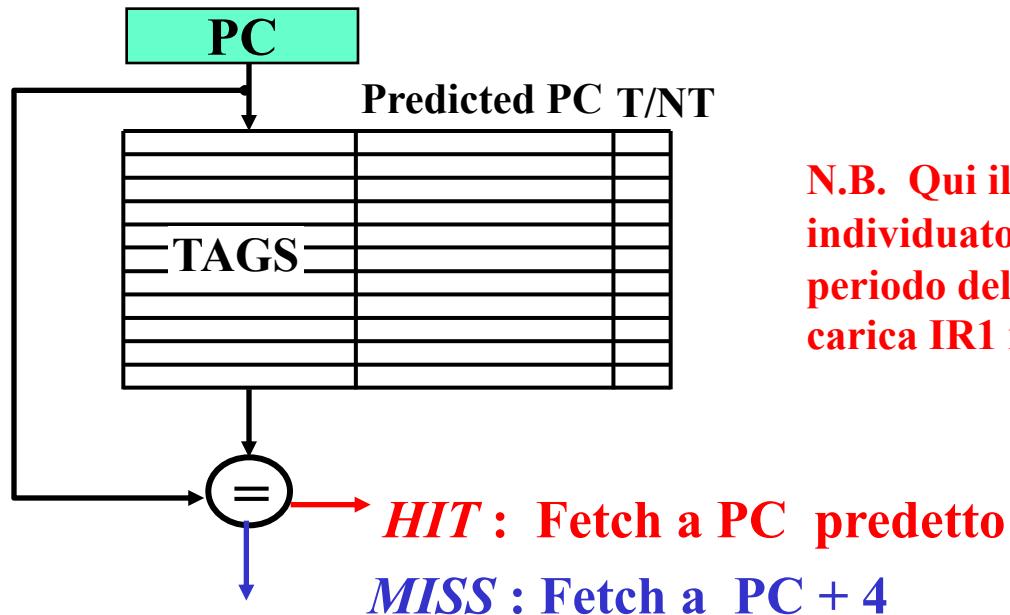
# Delayed branch/jump



*Al posto di una o più istruzioni “posposte” il compilatore mette delle NOP in caso non riesca a trovarne di adatte*

# Gestione delle Alee di Controllo con BTB

**Dynamic Prediction:** Branch Target Buffer -> nessuno stallo (quasi)

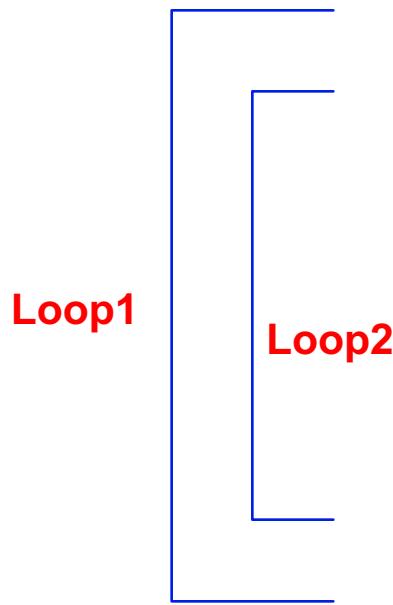


N.B. Qui il branch è individuato *durante* il periodo del clock IF che carica IR1 in IF/ID

Predizione Corretta : 0 stalli

Predizione Errata : da 1 a 3 stalli (fetch corretto in ID o EX v. precedentemente)

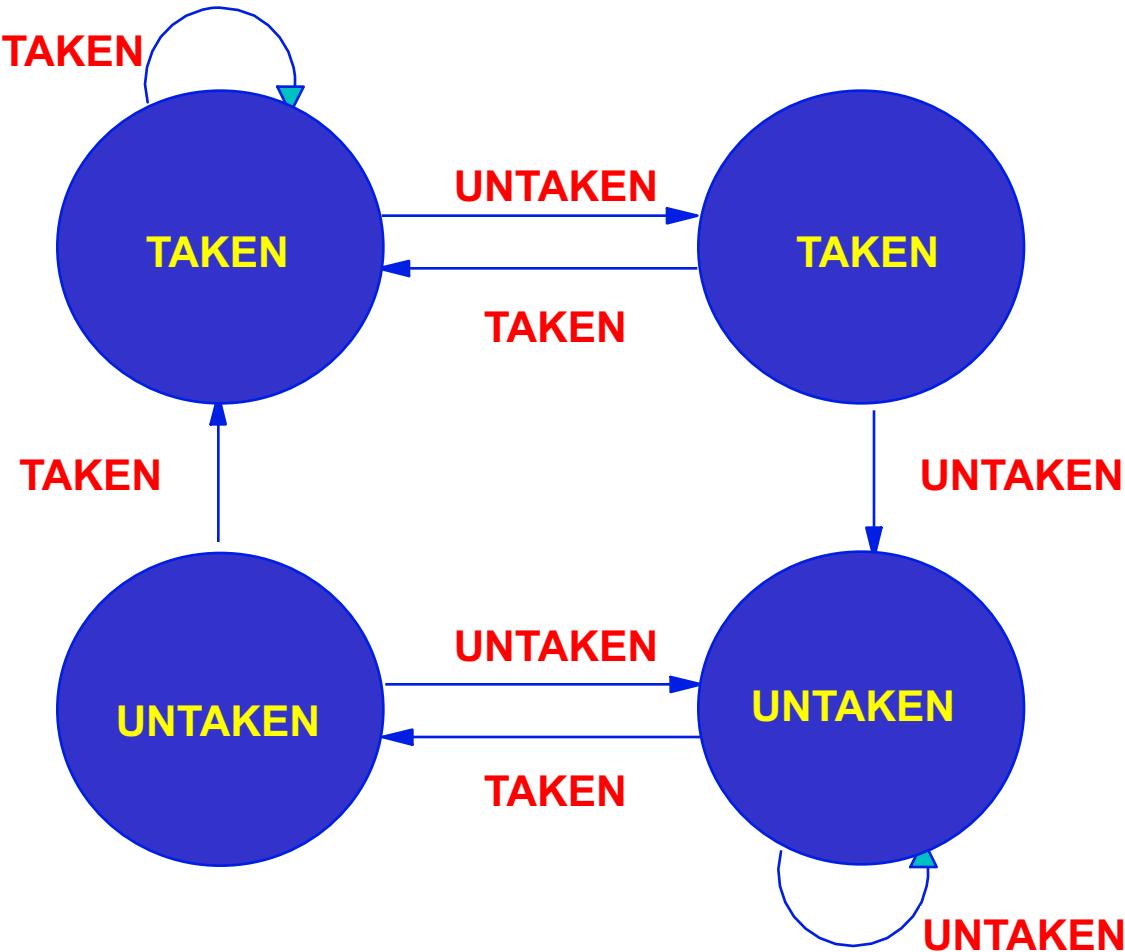
**Buffer di predizione: caso più semplice un bit che indica cosa è successo l'ultima volta.**



**Quando esce da loop2 sbaglia  
(predetto taken ma in realtà untaken)  
ma sbagli ancora quando predice  
untaken rientrando nuovamente in  
loop2 a causa di loop1**

**In presenza di preponderanza di un caso quando si verifica il caso opposto si hanno *due* errori successivi.**

Normalmente *due* bits.



**Esempio, molto frequente, di loop annidato:**

```
for (i=0; i<5000; i++)
    for (j=0; j<1000; j++)
    {
        x[i,j] = i*j + i + j;
        ...
        ...
    }
```